**Meta classes - Classes as objects**

Before understanding metaclasses, you need to master classes in Python. And Python has a very peculiar idea of what classes are, borrowed from the Smalltalk language.

In most languages, classes are just pieces of code that describe how to produce an object. That's kinda true in Python too:

>>> class ObjectCreator(object):

... pass

...

>>> my\_object = ObjectCreator()

>>> print my\_object

<\_\_main\_\_.ObjectCreator object at 0x8974f2c>

But classes are more than that in Python. Classes are objects too.

Yes, objects.

As soon as you use the keyword class, Python executes it and creates an OBJECT. The instruction

>>> class ObjectCreator(object):

... pass

...

creates in memory an object with the name ObjectCreator.

**This object (the class) is itself capable of creating objects (the instances), and this is why it's a class**.

But still, it's an object, and therefore:

* you can assign it to a variable
* you can copy it
* you can add attributes to it
* you can pass it as a function parameter

e.g.:

>>> print ObjectCreator # you can print a class because it's an object

<class '\_\_main\_\_.ObjectCreator'>

>>> def echo(o):

... print o

...

>>> echo(ObjectCreator) # you can pass a class as a parameter

<class '\_\_main\_\_.ObjectCreator'>

>>> print hasattr(ObjectCreator, 'new\_attribute')

False

>>> ObjectCreator.new\_attribute = 'foo' # you can add attributes to a class

>>> print hasattr(ObjectCreator, 'new\_attribute')

True

>>> print ObjectCreator.new\_attribute

foo

>>> ObjectCreatorMirror = ObjectCreator # you can assign a class to a variable

>>> print ObjectCreatorMirror.new\_attribute

foo

>>> print ObjectCreatorMirror()

<\_\_main\_\_.ObjectCreator object at 0x8997b4c>

**Creating classes dynamically**

Since classes are objects, you can create them on the fly, like any object.

First, you can create a class in a function using class:

>>> def choose\_class(name):

... if name == 'foo':

... class Foo(object):

... pass

... return Foo # return the class, not an instance

... else:

... class Bar(object):

... pass

... return Bar

...

>>> MyClass = choose\_class('foo')

>>> print MyClass # the function returns a class, not an instance

<class '\_\_main\_\_.Foo'>

>>> print MyClass() # you can create an object from this class

<\_\_main\_\_.Foo object at 0x89c6d4c>

But it's not so dynamic, since you still have to write the whole class yourself.

Since classes are objects, they must be generated by something.

When you use the class keyword, Python creates this object automatically. But as with most things in Python, it gives you a way to do it manually.

Remember the function type? The good old function that lets you know what type an object is:

>>> print type(1)

<type 'int'>

>>> print type("1")

<type 'str'>

>>> print type(ObjectCreator)

<type 'type'>

>>> print type(ObjectCreator())

<class '\_\_main\_\_.ObjectCreator'>

Well, type has a completely different ability, it can also create classes on the fly. type can take the description of a class as parameters, and return a class.

(I know, it's silly that the same function can have two completely different uses according to the parameters you pass to it. It's an issue due to backwards compatibility in Python)

type works this way:

type(name of the class,

tuple of the parent class (for inheritance, can be empty),

dictionary containing attributes names and values)

e.g.:

>>> class MyShinyClass(object):

... pass

can be created manually this way:

>>> MyShinyClass = type('MyShinyClass', (), {}) # returns a class object

>>> print MyShinyClass

<class '\_\_main\_\_.MyShinyClass'>

>>> print MyShinyClass() # create an instance with the class

<\_\_main\_\_.MyShinyClass object at 0x8997cec>

You'll notice that we use "MyShinyClass" as the name of the class and as the variable to hold the class reference. They can be different, but there is no reason to complicate things.

type accepts a dictionary to define the attributes of the class. So:

>>> class Foo(object):

... bar = True

Can be translated to:

>>> Foo = type('Foo', (), {'bar':True})

And used as a normal class:

>>> print Foo

<class '\_\_main\_\_.Foo'>

>>> print Foo.bar

True

>>> f = Foo()

>>> print f

<\_\_main\_\_.Foo object at 0x8a9b84c>

>>> print f.bar

True

And of course, you can inherit from it, so:

>>> class FooChild(Foo):

... pass

would be:

>>> FooChild = type('FooChild', (Foo,), {})

>>> print FooChild

<class '\_\_main\_\_.FooChild'>

>>> print FooChild.bar # bar is inherited from Foo

True

Eventually you'll want to add methods to your class. Just define a function with the proper signature and assign it as an attribute.

>>> def echo\_bar(self):

... print self.bar

...

>>> FooChild = type('FooChild', (Foo,), {'echo\_bar': echo\_bar})

>>> hasattr(Foo, 'echo\_bar')

False

>>> hasattr(FooChild, 'echo\_bar')

True

>>> my\_foo = FooChild()

>>> my\_foo.echo\_bar()

True

You see where we are going: in Python, classes are objects, and you can create a class on the fly, dynamically.

This is what Python does when you use the keyword class, and it does so by using a metaclass.

**What are metaclasses (finally)**

Metaclasses are the 'stuff' that creates classes.

You define classes in order to create objects, right?

But we learned that Python classes are objects.

Well, metaclasses are what create these objects. They are the classes' classes, you can picture them this way:

MyClass = MetaClass()

MyObject = MyClass()

You've seen that type lets you do something like this:

MyClass = type('MyClass', (), {})

It's because the function type is in fact a metaclass. type is the metaclass Python uses to create all classes behind the scenes.

Now you wonder why the heck is it written in lowercase, and not Type?

Well, I guess it's a matter of consistency with str, the class that creates strings objects, and int the class that creates integer objects. type is just the class that creates class objects.

You see that by checking the \_\_class\_\_ attribute.

Everything, and I mean everything, is an object in Python. That includes ints, strings, functions and classes. All of them are objects. And all of them have been created from a class:

>>> age = 35

>>> age.\_\_class\_\_

<type 'int'>

>>> name = 'bob'

>>> name.\_\_class\_\_

<type 'str'>

>>> def foo(): pass

>>> foo.\_\_class\_\_

<type 'function'>

>>> class Bar(object): pass

>>> b = Bar()

>>> b.\_\_class\_\_

<class '\_\_main\_\_.Bar'>

Now, what is the \_\_class\_\_ of any \_\_class\_\_ ?

>>> age.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> name.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> foo.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> b.\_\_class\_\_.\_\_class\_\_

<type 'type'>

So, a metaclass is just the stuff that creates class objects.

You can call it a 'class factory' if you wish.

type is the built-in metaclass Python uses, but of course, you can create your own metaclass.

**The \_\_metaclass\_\_ attribute**

You can add a \_\_metaclass\_\_ attribute when you write a class:

class Foo(object):

\_\_metaclass\_\_ = something...

[...]

If you do so, Python will use the metaclass to create the class Foo.

Careful, it's tricky.

You write class Foo(object) first, but the class object Foo is not created in memory yet.

Python will look for \_\_metaclass\_\_ in the class definition. If it finds it, it will use it to create the object class Foo. If it doesn't, it will use type to create the class.

Read that several times.

When you do:

class Foo(Bar):

pass

Python does the following:

Is there a \_\_metaclass\_\_ attribute in Foo?

If yes, create in memory a class object (I said a class object, stay with me here), with the name Foo by using what is in \_\_metaclass\_\_.

If Python can't find \_\_metaclass\_\_, it will look for a \_\_metaclass\_\_ in Bar (the parent class), and try to do the same.

If Python can't find \_\_metaclass\_\_ in any parent, it will look for a \_\_metaclass\_\_ at the MODULE level, and try to do the same.

Then if it can't find any \_\_metaclass\_\_ at all, it will use type to create the class object.

Now the big question is, what can you put in \_\_metaclass\_\_ ?

The answer is: something that can create a class.

And what can create a class? type, or anything that subclasses or uses it.

**Custom metaclasses**

The main purpose of a metaclass is to change the class automatically, when it's created.

You usually do this for APIs, where you want to create classes matching the current context.

Imagine a stupid example, where you decide that all classes in your module should have their attributes written in uppercase. There are several ways to do this, but one way is to set \_\_metaclass\_\_ at the module level.

This way, all classes of this module will be created using this metaclass, and we just have to tell the metaclass to turn all attributes to uppercase.

Luckily, \_\_metaclass\_\_ can actually be any callable, it doesn't need to be a formal class (I know, something with 'class' in its name doesn't need to be a class, go figure... but it's helpful).

So we will start with a simple example, by using a function.

# the metaclass will automatically get passed the same argument

# that you usually pass to `type`

def upper\_attr(future\_class\_name, future\_class\_parents, future\_class\_attr):

"""

Return a class object, with the list of its attribute turned

into uppercase.

"""

# pick up any attribute that doesn't start with '\_\_' and uppercase it

uppercase\_attr = {}

for name, val in future\_class\_attr.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

# let `type` do the class creation

return type(future\_class\_name, future\_class\_parents, uppercase\_attr)

\_\_metaclass\_\_ = upper\_attr # this will affect all classes in the module

class Foo(): # global \_\_metaclass\_\_ won't work with "object" though

# but we can define \_\_metaclass\_\_ here instead to affect only this class

# and this will work with "object" children

bar = 'bip'

print hasattr(Foo, 'bar')

# Out: False

print hasattr(Foo, 'BAR')

# Out: True

f = Foo()

print f.BAR

# Out: 'bip'

Now, let's do exactly the same, but using a real class for a metaclass:

# remember that `type` is actually a class like `str` and `int`

# so you can inherit from it

class UpperAttrMetaclass(type):

# \_\_new\_\_ is the method called before \_\_init\_\_

# it's the method that creates the object and returns it

# while \_\_init\_\_ just initializes the object passed as parameter

# you rarely use \_\_new\_\_, except when you want to control how the object

# is created.

# here the created object is the class, and we want to customize it

# so we override \_\_new\_\_

# you can do some stuff in \_\_init\_\_ too if you wish

# some advanced use involves overriding \_\_call\_\_ as well, but we won't

# see this

def \_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, future\_class\_attr):

uppercase\_attr = {}

for name, val in future\_class\_attr.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

return type(future\_class\_name, future\_class\_parents, uppercase\_attr)

But this is not really OOP. We call type directly and we don't override call the parent \_\_new\_\_. Let's do it:

class UpperAttrMetaclass(type):

def \_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, future\_class\_attr):

uppercase\_attr = {}

for name, val in future\_class\_attr.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

# reuse the type.\_\_new\_\_ method

# this is basic OOP, nothing magic in there

return type.\_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, uppercase\_attr)

You may have noticed the extra argument upperattr\_metaclass. There is nothing special about it: a method always receives the current instance as first parameter. Just like you have self for ordinary methods.

Of course, the names I used here are long for the sake of clarity, but like for self, all the arguments have conventional names. So a real production metaclass would look like this:

class UpperAttrMetaclass(type):

def \_\_new\_\_(cls, clsname, bases, dct):

uppercase\_attr = {}

for name, val in dct.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

return type.\_\_new\_\_(cls, clsname, bases, uppercase\_attr)

We can make it even cleaner by using super, which will ease inheritance (because yes, you can have metaclasses, inheriting from metaclasses, inheriting from type):

class UpperAttrMetaclass(type):

def \_\_new\_\_(cls, clsname, bases, dct):

uppercase\_attr = {}

for name, val in dct.items():

if not name.startswith('\_\_'):

uppercase\_attr[name.upper()] = val

else:

uppercase\_attr[name] = val

return super(UpperAttrMetaclass, cls).\_\_new\_\_(cls, clsname, bases, uppercase\_attr)

That's it. There is really nothing more about metaclasses.

The reason behind the complexity of the code using metaclasses is not because of metaclasses, it's because you usually use metaclasses to do twisted stuff relying on introspection, manipulating inheritance, vars such as \_\_dict\_\_, etc.

Indeed, metaclasses are especially useful to do black magic, and therefore complicated stuff. But by themselves, they are simple:

* intercept a class creation
* modify the class
* return the modified class

**Why would you use metaclasses classes instead of functions?**

Since \_\_metaclass\_\_ can accept any callable, why would you use a class since it's obviously more complicated?

There are several reasons to do so:

* The intention is clear. When you read UpperAttrMetaclass(type), you know what's going to follow
* You can use OOP. Metaclass can inherit from metaclass, override parent methods. Metaclasses can even use metaclasses.
* You can structure your code better. You never use metaclasses for something as trivial as the above example. It's usually for something complicated. Having the ability to make several methods and group them in one class is very useful to make the code easier to read.
* You can hook on \_\_new\_\_, \_\_init\_\_ and \_\_call\_\_. Which will allow you to do different stuff. Even if usually you can do it all in \_\_new\_\_, some people are just more comfortable using \_\_init\_\_.
* These are called metaclasses, damn it! It must mean something!

**Why the hell would you use metaclasses?**

Now the big question. Why would you use some obscure error prone feature?

Well, usually you don't:

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).

*Python Guru Tim Peters*

The main use case for a metaclass is creating an API. A typical example of this is the Django ORM.

It allows you to define something like this:

class Person(models.Model):

name = models.CharField(max\_length=30)

age = models.IntegerField()

But if you do this:

guy = Person(name='bob', age='35')

print guy.age

It won't return an IntegerField object. It will return an int, and can even take it directly from the database.

This is possible because models.Model defines \_\_metaclass\_\_ and it uses some magic that will turn the Person you just defined with simple statements into a complex hook to a database field.

Django makes something complex look simple by exposing a simple API and using metaclasses, recreating code from this API to do the real job behind the scenes.

**The last word**

First, you know that classes are objects that can create instances.

Well in fact, classes are themselves instances. Of metaclasses.

>>> class Foo(object): pass

>>> id(Foo)

142630324

Everything is an object in Python, and they are all either instances of classes or instances of metaclasses.

Except for type.

type is actually its own metaclass. This is not something you could reproduce in pure Python, and is done by cheating a little bit at the implementation level.

Secondly, metaclasses are complicated. You may not want to use them for very simple class alterations. You can change classes by using two different techniques:

* monkey patching
* class decorators

99% of the time you need class alteration, you are better off using these.

But 99% of the time, you don't need class alteration at all :-)

**Second Post**

A metaclass is the class of a class. Like a class defines how an instance of the class behaves, a metaclass defines how a class behaves. A class is an instance of a metaclass.

While in Python you can use arbitrary callables for metaclasses (like [Jerub](http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python/100037" \l "100037) shows), the more useful approach is actually to make it an actual class itself. 'type' is the usual metaclass in Python. In case you're wondering, yes, 'type' is itself a class, and it is its own type. You won't be able to recreate something like 'type' purely in Python, but Python cheats a little. To create your own metaclass in Python you really just want to subclass 'type'.

A metaclass is most commonly used as a class-factory. Like you create an instance of the class by calling the class, Python creates a new class (when it executes the 'class' statement) by calling the metaclass. Combined with the normal \_\_init\_\_ and \_\_new\_\_ methods, metaclasses therefor allow you to do 'extra things' when creating a class, like registering the new class with some registry, or even replace the class with something else entirely.

When the 'class' statement is executed, Python first executes the body of the 'class' statement as a normal block of code. The resulting namespace (a dict) holds the attributes of the class-to-be. The metaclass is determined by looking at the baseclasses of the class-to-be (metaclasses are inherited), at the \_\_metaclass\_\_ attribute of the class-to-be (if any) or the '\_\_metaclass\_\_' global variable. The metaclass is then called with the name, bases and attributes of the class to instantiate it.

However, metaclasses actually define the *type* of a class, not just a factory for it, so you can do much more with them. You can, for instance, define normal methods on the metaclass. These metaclass-methods are like classmethods, in that they can be called on the class without an instance, but they are also not like classmethods in that they cannot be called on an instance of the class. type.\_\_subclasses\_\_() is an example of a method on the 'type' metaclass. You can also define the normal 'magic' methods, like \_\_add\_\_, \_\_iter\_\_ and \_\_getattr\_\_, to implement or change how the class behaves.

Here's an aggregated example of the bits and pieces:

def make\_hook(f):

"""Decorator to turn 'foo' method into '\_\_foo\_\_'"""

f.is\_hook = 1

return f

class MyType(type):

def \_\_new\_\_(cls, name, bases, attrs):

if name.startswith('None'):

return None

# Go over attributes and see if they should be renamed.

newattrs = {}

for attrname, attrvalue in attrs.iteritems():

if getattr(attrvalue, 'is\_hook', 0):

newattrs['\_\_%s\_\_' % attrname] = attrvalue

else:

newattrs[attrname] = attrvalue

return super(MyType, cls).\_\_new\_\_(cls, name, bases, newattrs)

def \_\_init\_\_(self, name, bases, attrs):

super(MyType, self).\_\_init\_\_(name, bases, attrs)

# classregistry.register(self, self.interfaces)

print "Would register class %s now." % self

def \_\_add\_\_(self, other):

class AutoClass(self, other):

pass

return AutoClass

# Alternatively, to autogenerate the classname as well as the class:

# return type(self.\_\_name\_\_ + other.\_\_name\_\_, (self, other), {})

def unregister(self):

# classregistry.unregister(self)

print "Would unregister class %s now." % self

class MyObject:

\_\_metaclass\_\_ = MyType

class NoneSample(MyObject):

pass

# Will print "NoneType None"

print type(NoneSample), repr(NoneSample)

class Example(MyObject):

def \_\_init\_\_(self, value):

self.value = value

@make\_hook

def add(self, other):

return self.\_\_class\_\_(self.value + other.value)

# Will unregister the class

Example.unregister()

inst = Example(10)

# Will fail with an AttributeError

#inst.unregister()

print inst + inst

class Sibling(MyObject):

pass

ExampleSibling = Example + Sibling

# ExampleSibling is now a subclass of both Example and Sibling (with no

# content of its own) although it will believe it's called 'AutoClass'

print ExampleSibling

print ExampleSibling.\_\_mro\_\_

**Third Post**

Metaclasses are the secret sauce that make 'class' work. The default metaclass for a new style object is called 'type'.

class type(object)

| type(object) -> the object's type

| type(name, bases, dict) -> a new type

Metaclasses take 3 args. '**name**', '**bases**' and '**dict**'

Here is where the secret starts. Look for where name, bases and the dict come from in this example class definition.

class ThisIsTheName(Bases, Are, Here):

All\_the\_code\_here

def doesIs(create, a):

dict

Lets define a metaclass that will demonstrate how '**class:**' calls it.

def test\_metaclass(name, bases, dict):

print 'The Class Name is', name

print 'The Class Bases are', bases

print 'The dict has', len(dict), 'elems, the keys are', dict.keys()

return "yellow"

class TestName(object, None, int, 1):

\_\_metaclass\_\_ = test\_metaclass

foo = 1

def baz(self, arr):

pass

print 'TestName = ', repr(TestName)

# output =>

The Class Name is TestName

The Class Bases are (<type 'object'>, None, <type 'int'>, 1)

The dict has 4 elems, the keys are ['baz', '\_\_module\_\_', 'foo', '\_\_metaclass\_\_']

TestName = 'yellow'

And now, an example that actually means something, this will automatically make the variables in the list "attributes" set on the class, and set to None.

def init\_attributes(name, bases, dict):

if 'attributes' in dict:

for attr in dict['attributes']:

dict[attr] = None

return type(name, bases, dict)

class Initialised(object):

\_\_metaclass\_\_ = init\_attributes

attributes = ['foo', 'bar', 'baz']

print 'foo =>', Initialised.foo

# output=>

foo => None

Note that the magic behaviour that 'Initalised' gains by having the metaclass init\_attributes is not passed onto a subclass of Initalised.

Here is an even more concrete example, showing how you can subclass 'type' to make a metaclass that performs an action when the class is created. This is quite tricky:

class MetaSingleton(type):

instance = None

def \_\_call\_\_(cls, \*args, \*\*kw):

if cls.instance is None:

cls.instance = super(MetaSingleton, cls).\_\_call\_\_(\*args, \*\*kw)

return cls.instance

class Foo(object):

\_\_metaclass\_\_ = MetaSingleton

a = Foo()

b = Foo()

assert a is b

# Unifying types and classes in Python 2.2

### Table of Contents

* [Introduction](http://www.python.org/download/releases/2.2/descrintro/" \l "introduction)
* [Subclassing built-in types](http://www.python.org/download/releases/2.2/descrintro/" \l "subclassing)
* [Built-in types as factory functions](http://www.python.org/download/releases/2.2/descrintro/" \l "factories)
* [Introspecting instances of built-in types](http://www.python.org/download/releases/2.2/descrintro/" \l "introspection)
* [Static methods and class methods](http://www.python.org/download/releases/2.2/descrintro/" \l "staticmethods)
* [Properties: attributes defined by get/set methods](http://www.python.org/download/releases/2.2/descrintro/" \l "property)
* [Method resolution order](http://www.python.org/download/releases/2.2/descrintro/" \l "mro)
* [Cooperative methods and "super"](http://www.python.org/download/releases/2.2/descrintro/" \l "cooperation)
* [Overriding the \_\_new\_\_ method](http://www.python.org/download/releases/2.2/descrintro/" \l "__new__)
* [Metaclasses](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclasses)
* [Backwards incompatibilities](http://www.python.org/download/releases/2.2/descrintro/" \l "incompatibilities)
* [References](http://www.python.org/download/releases/2.2/descrintro/" \l "references)

### Introduction

Python 2.2 introduces the first phase of "type/class unification". This is a series of changes to Python intended to remove most of the differences between built-in types and user-defined classes. Perhaps the most obvious one is the restriction against using built-in types (such as the type of lists and dictionaries) as a base class in a class statement.

This is one of the biggest changes to Python ever, and yet it can be done with very few backwards incompatibilities. The changes are described in minute detail in a series of [PEPs](http://www.python.org/download/releases/2.2/descrintro/" \l "references) (Python Enhancement Proposals). PEPs are not designed to be tutorials, and the PEPs describing the type/class unification are sometimes hard to read. They also aren't finished yet. That's where this paper comes in: it introduces the key elements of the type/class unification for the average Python programmer.

A bit of terminology: "classic Python" refers to Python 2.1 (and its patch releases such as 2.1.1) or earlier versions, while "classic classes" refer to classes defined with a class statement that does not have a built-in object amongst its bases: either because it has no bases, or because all of its bases are classic classes themselves - applying the definition recursively.

Classic classes are still a special category in Python 2.2. Eventually they will be totally unified with types, but because of additional backwards incompatibilities, this will be done after 2.2 is released (maybe not before Python 3.0). I'll try to say "type" when I mean a built-in type, and "class" when I'm referring to a classic class or something that could be either; if it wouldn't be clear from the context which interpretation is meant, I'll try to be explicit, using "classic class" or "class or type".

### Subclassing built-in types

Let's start with the juiciest bit: you can subtype built-in types like dictionaries and lists. All you need is a name for a base class that is a built-in type and you're in business.

There's a new built-in name, "dict", for the type of dictionaries. (In version 2.2b1 and before, this was called "dictionary"; while in general I don't like abbreviations, "dictionary" was just too long to type, and we've been saying "dict" for years.)

This is really just sugar, since there are already two other ways to name this type: type({}) and (after importing the types module) types.DictType (and a third, types.DictionaryType). But now that types play a more central role, it seems appropriate to have built-in names for the types that you're likely to encounter.

Here's an example of a simple dict subclass, which provides a "default value" that is returned when a missing key is requested:

class defaultdict(dict):

def \_\_init\_\_(self, default=None):

dict.\_\_init\_\_(self)

self.default = default

def \_\_getitem\_\_(self, key):

try:

return dict.\_\_getitem\_\_(self, key)

except KeyError:

return self.default

This example shows a few things. The \_\_init\_\_() method extends the dict.\_\_init\_\_() method. Like \_\_init\_\_() methods are wont to do, it has a different argument list than the base class \_\_init\_\_() method. Likewise, the \_\_getitem\_\_() method extends the base class \_\_getitem\_\_() method.

The \_\_getitem\_\_() method could also be written as follows, using the new "key in dict" test introduced in Python 2.2:

def \_\_getitem\_\_(self, key):

if key in self:

return dict.\_\_getitem\_\_(self, key)

else:

return self.default

I believe that this version is less efficient, because it does the key lookup twice. The exception would be when we expect that the requested key is almost never in the dictionary: then setting up the try/except statement is more expensive than the failing "key in self" test.

To be complete, the get() method should probably also be extended, to make it use the same default as \_\_getitem\_\_():

def get(self, key, \*args):

if not args:

args = (self.default,)

return dict.get(self, key, \*args)

(Although this function is declared with a variable-length argument list, it really should only be called with one or two arguments; if more are passed, the base class method call will raise a TypeError exception.)

We're not restricted to extending methods defined on the base class. Here's a useful method that does something similar to update(), but keeps existing values rather than overwriting them with new values if a key exists in both dictionaries:

def merge(self, other):

for key in other:

if key not in self:

self[key] = other[key]

This uses the new "key not in dict" test as well as the new "for key in dict:" to iterate efficiently (without making a copy of the list of keys) over all keys in a dictionary. It doesn't require the other argument to be a defaultdict or even a dictionary: any mapping object that supports "for key in other" and other[key] will do.

Here's the new type at work:

>>> print defaultdict # show our type

<class '\_\_main\_\_.defaultdict'>

>>> print type(defaultdict) # its metatype

<type 'type'>

>>> a = defaultdict(default=0.0) # create an instance

>>> print a # show the instance

{}

>>> print type(a) # show its type

<class '\_\_main\_\_.defaultdict'>

>>> print a.\_\_class\_\_ # show its class

<class '\_\_main\_\_.defaultdict'>

>>> print type(a) is a.\_\_class\_\_ # its type is its class

1

>>> a[1] = 3.25 # modify the instance

>>> print a # show the new value

{1: 3.25}

>>> print a[1] # show the new item

3.25

>>> print a[0] # a non-existant item

0.0

>>> a.merge({1:100, 2:200}) # use a dictionary method

>>> print a # show the result

{1: 3.25, 2: 200}

>>>

We can also use the new type in contexts where classic only allows "real" dictionaries, such as the locals/globals dictionaries for the exec statement or the built-in function eval():

>>> print a.keys()

[1, 2]

>>> exec "x = 3; print x" in a

3

>>> print a.keys()

['\_\_builtins\_\_', 1, 2, 'x']

>>> print a['x']

3

>>>

However, our \_\_getitem\_\_() method is not used for variable access by the interpreter:

>>> exec "print foo" in a

Traceback (most recent call last):

File "<stdin>", line 1, in ?

File "<string>", line 1, in ?

NameError: name 'foo' is not defined

>>>

Why doesn't this print 0.0? The interpreter uses an internal function to access the dictionary, which bypasses our \_\_getitem\_\_() override. I admit that this can be a problem (although it is *only* a problem in this context, when a dict subclass is used as a locals/globals dictionary); it remains to be seen if I can fix this without compromising performance in the common case.

Now we'll see that defaultdict instances have dynamic instance variables, just like classic classes:

>>> a.default = -1

>>> print a["noway"]

-1

>>> a.default = -1000

>>> print a["noway"]

-1000

>>> print a.\_\_dict\_\_.keys()

['default']

>>> a.x1 = 100

>>> a.x2 = 200

>>> print a.x1

100

>>> print a.\_\_dict\_\_.keys()

['default', 'x2', 'x1']

>>> print a.\_\_dict\_\_

{'default': -1000, 'x2': 200, 'x1': 100}

>>>

This is not always what you want; in particular, using a separate dictionary to hold a single instance variable doubles the memory used by a defaultdict instance compared to using a regular dictionary! There's a way to avoid this:

class defaultdict2(dict):

\_\_slots\_\_ = ['default']

def \_\_init\_\_(self, default=None):

*...(like before)...*

The \_\_slots\_\_ declaration takes a list of instance variables, and reserves space in the instance for exactly these in the instance. When \_\_slots\_\_ is used, other instance variables cannot be assigned to:

>>> a = defaultdict2(default=0.0)

>>> a[1]

0.0

>>> a.default = -1

>>> a[1]

-1

>>> a.x1 = 1

Traceback (most recent call last):

File "<stdin>", line 1, in ?

AttributeError: 'defaultdict2' object has no attribute 'x1'

>>>

Some noteworthy tidbits and warnings about \_\_slots\_\_:

* An undefined slot variable will raise AttributeError as expected. (Note that in Python 2.2b2 and earlier, slot variables had the value None by default, and "deleting" them restores this default value.)
* You cannot use a class attribute to define a default value for an instance variable defined by \_\_slots\_\_; the \_\_slots\_\_ declaration creates a class attribute for each variable containing a descriptor, and setting a class attribute to a default value would overwrite this descriptor.
* There's no check that prevents you to override an instance variable already defined by a base class using a \_\_slots\_\_ declaration. If you do that, the instance variable defined by the base class is inaccessible (except by retrieving its descriptor directly from the base class; this could be used to rename it). Doing this renders the meaning of your program undefined; a check to prevent this may be added in the future.
* Instances of a class that uses \_\_slots\_\_ don't have a \_\_dict\_\_ (unless a base class defines a \_\_dict\_\_); but instances of derived classes of it do have a \_\_dict\_\_, unless their class also uses \_\_slots\_\_.
* You can define an object with no instance variables and no \_\_dict\_\_ by using \_\_slots\_\_ = [].
* You cannot use slots with "variable-length" built-in types as base class. Variable-length built-in types are long, str and tuple.
* A class using \_\_slots\_\_ does not support weak references to its instances, unless one of the strings in the \_\_slots\_\_ list equals "\_\_weakref\_\_". (Hmm, this feature could be extended to "\_\_dict\_\_"...)
* The \_\_slots\_\_ variable doesn't have to be a list; any non-string that can be iterated over will do, and the values returned by the iteration are used as the slot names. In particular, a dictionary can be used. You can also use a single string, to declare a single slot. However, in the future, an additional meaning may be assigned to using a dictionary, for example, the dictionary values may be used to restrict the type of an instance variable or provide a doc string; the effect of using something that's not a list renders the meaning of your program undefined.

Note that while in general operator overloading works just as for classic classes, there are some differences. (The biggest one is the lack of support for \_\_coerce\_\_; new-style classes should always use the new-style numeric API, which passes the other operand uncoerced to the \_\_add\_\_ and \_\_radd\_\_ methods, etc.)

There's a new way of overriding attribute access. The \_\_getattr\_\_ hook, if defined, works the same way as it does for classic classes: it is only called if the regular way of searching for the attribute doesn't find it. But you can now also override \_\_getattribute\_\_, a new operation that is called for *all* attribute references.

When overriding \_\_getattribute\_\_, bear in mind that it is easy to cause infinite recursion: whenever \_\_getattribute\_\_ references an attribute of self (even self.\_\_dict\_\_!), it is called recursively. (This is similar to \_\_setattr\_\_, which gets called for all attribute assignments; \_\_getattr\_\_ can also suffer from this when it is carelessly written and references a non-existent attribute of self.)

The correct way to get any attribute from self inside \_\_getattribute\_\_ is to call the base class's \_\_getattribute\_\_ method, in the same way any method that overrides a base class method can call the base class method: Base.\_\_getattribute\_\_(self, name). (See also the discussion of [super()](http://www.python.org/download/releases/2.2/descrintro/" \l "cooperation) below if you want to be correct in a multiple inheritance world.)

Here's an example of overriding \_\_getattribute\_\_ (really extending it, since the overriding method calls the base class method):

class C(object):

def \_\_getattribute\_\_(self, name):

print "accessing %r.%s" % (self, name)

return object.\_\_getattribute\_\_(self, name)

A note about \_\_setattr\_\_: sometimes attributes are not stored in self.\_\_dict\_\_ (for example when using \_\_slots\_\_ or properties, or when using a built-in base class). The same pattern as for \_\_getattribute\_\_ applies, where you call the base class \_\_setattr\_\_ to do the actual work. Here's an example:

class C(object):

def \_\_setattr\_\_(self, name, value):

if hasattr(self, name):

raise AttributeError, "attributes are write-once"

object.\_\_setattr\_\_(self, name, value)

C++ programmers may find it useful to realize that this form of subtyping in Python is implemented very similarly to single-inheritance subclassing in C++, with \_\_class\_\_ in the role of the vtable.

There's much more that could be explained (like the \_\_metaclass\_\_ declaration, and the \_\_new\_\_ method), but most of that is pretty esoteric. See [below](http://www.python.org/download/releases/2.2/descrintro/" \l "__new__) if you're interested.

I'll end with a list of caveats:

* You can use multiple inheritance, but you can't multiply inherit from different built-in types (for example, you can't create a type that inherits from both the built-in dict and list types). This is a permanent restriction; it would require too many changes to Python's object implementation to lift it. However, you can create mix-in classes by inheriting from "object". This is a new built-in, naming the featureless base type of all built-in types under the new system.
* When using multiple inheritance, you can mix classic classes and built-in types (or types derived from built-in types) in the list of base classes. (This is new in Python 2.2b2; in earlier versions you couldn't.)
* See also the general [bugs in 2.2 list](http://www.python.org/download/releases/2.2/descrintro/bugs).

### Built-in types as factory functions

The previous section showed that an instance of the built-in subtype defaultdict can be created by calling defaultdict(). This is expected, because this also works for classic classes. But here's a new feature: built-in base types themselves can also be instantiated by calling the type directly.

For several built-in types, there are already factory functions named after the type in classic Python, for example str() and int(). I've changed these built-ins so that they are now names for the corresponding types. While this changes the type of these names from built-in function to built-in type, I don't expect that this will create backward compatibility problems: I've made sure that the types can be called with exactly the same argument lists as the former functions. (They can also generally be called without arguments, producing an object with a suitable default value, such as zero or empty; this is new.)

These are the affected built-ins:

* int([number\_or\_string[, base\_number]])
* long([number\_or\_string])
* float([number\_or\_string])
* complex([number\_or\_string[, imag\_number]])
* str([object])
* unicode([string[, encoding\_string]])
* tuple([iterable])
* list([iterable])
* type(object) or type(name\_string, bases\_tuple, methods\_dict)

The signature of type() requires an explanation: traditionally, type(x) returns the type of object x, and this usage is still supported. However, type(name, bases, methods) is a new usage that creates a brand new type object. (This gets into [metaclass programming](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclasses), and I won't go into this further here except to note that this signature is the same as that used by the Don Beaudry hook of metaclass fame.)

There are also a few new built-ins that follow the same pattern. These have been described above or will be described below:

* dict([mapping\_or\_iterable]) - return a new dictionary; the optional argument must be either a mapping whose items are copied, or a sequence of 2-tuples of length 2 giving the (key, value) pairs to be inserted into the new dictionary
* object([...]) - return a new featureless object; arguments are ignored
* classmethod(function) - see [below](http://www.python.org/download/releases/2.2/descrintro/" \l "staticmethods)
* staticmethod(function) - see [below](http://www.python.org/download/releases/2.2/descrintro/" \l "staticmethods)
* super(class\_or\_type[, instance]) - see [below](http://www.python.org/download/releases/2.2/descrintro/" \l "cooperation)
* property([fget[, fset[, fdel[, doc]]]]) - see [below](http://www.python.org/download/releases/2.2/descrintro/" \l "property)

The purpose of this change is twofold. First, this makes it convenient to use any of these types as a base class in a class statement. Second, it makes testing for a specific type a little easier: rather than writing type(x) is type(0), you can now write isinstance(x, int).

Which reminds me. The second argument of isinstance() may now be a tuple of classes or types. For example, isinstance(x, (int, long)) returns true when x is an int or a long (or an instance of a subclass of either of those types), and similarly isinstance(x, (str, unicode)) tests for a string of either variety. We didn't do this to isclass().

### Introspecting instances of built-in types

For instances of built-in types (and for new-style classes in general), x.\_\_class\_\_ is now the same as type(x):

>>> type([])

<type 'list'>

>>> [].\_\_class\_\_

<type 'list'>

>>> list

<type 'list'>

>>> isinstance([], list)

1

>>> isinstance([], dict)

0

>>> isinstance([], object)

1

>>>

In classic Python, the method names of lists were available as the \_\_methods\_\_ attribute of list objects, with the same effect as using the built-in dir() function:

Python 2.1 (#30, Apr 18 2001, 00:47:18)

[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2

Type "copyright", "credits" or "license" for more information.

>>> [].\_\_methods\_\_

['append', 'count', 'extend', 'index', 'insert', 'pop',

'remove', 'reverse', 'sort']

>>>

>>> dir([])

['append', 'count', 'extend', 'index', 'insert', 'pop',

'remove', 'reverse', 'sort']

Under the new proposal, the \_\_methods\_\_ attribute no longer exists:

Python 2.2c1 (#803, Dec 13 2001, 23:06:05)

[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2

Type "copyright", "credits" or "license" for more information.

>>> [].\_\_methods\_\_

Traceback (most recent call last):

File "<stdin>", line 1, in ?

AttributeError: 'list' object has no attribute '\_\_methods\_\_'

>>>

Instead, you can get the same information from the dir() function, which gives more information:

>>> dir([])

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_',

'\_\_delitem\_\_', '\_\_eq\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_',

'\_\_getitem\_\_', '\_\_getslice\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_',

'\_\_imul\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_',

'\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_repr\_\_', '\_\_rmul\_\_',

'\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_setslice\_\_', '\_\_str\_\_', 'append',

'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',

'sort']

>>>

The new dir() gives more information than the old one: in addition to the names of instance variables and regular methods, it also shows the methods that are normally invoked through special notations, like \_\_iadd\_\_ (+=), \_\_len\_\_ (len), \_\_ne\_\_ (!=).

More about the new dir() function:

* dir() on an instance (classic or new-style) shows the instance variables as well as the methods and class attributes defined by the instance's class and all its base classes.
* dir() on a class (classic or new-style) shows the contents of the \_\_dict\_\_ of the class and all its base classes. It does not show class attributes that are defined by a metaclass.
* dir() on a module shows the contents of the module's \_\_dict\_\_. (This is unchanged.)
* dir() without arguments shows the caller's local variables. (Again, unchanged.)
* There's a new C API that implements the dir() function: PyObject\_Dir().
* There are more details; in particular, for objects that override \_\_dict\_\_ or \_\_class\_\_, these are honored, and for backwards compatibility, \_\_members\_\_ and \_\_methods\_\_ are honored if they are defined.

You can use a method of a built-in type as an "unbound method":

>>> a = ['tic', 'tac']

>>> list.\_\_len\_\_(a) # same as len(a)

2

>>> list.append(a, 'toe') # same as a.append('toe')

>>> a

['tic', 'tac', 'toe']

>>>

This is just like using an unbound method of a user-defined class - and similarly, it's mostly useful from inside a subclass method, to call the corresponding base class method.

Unlike user-defined classes, you cannot change built-in types: attempts to assign an attribute of a built-in type raises a TypeError, and their \_\_dict\_\_ is a read-only proxy object. The restriction on attribute assignment is lifted for new-style user-defined classes, including subclasses of built-in types; however even those have a read-only \_\_dict\_\_ proxy, and you must use attribute assignment to replace or add a method of a new-style class. Example session:

>>> list.append

<method 'append' of 'list' objects>

>>> list.append = list.append

Traceback (most recent call last):

File "<stdin>", line 1, in ?

TypeError: can't set attributes of built-in/extension type 'list'

>>> list.answer = 42

Traceback (most recent call last):

File "<stdin>", line 1, in ?

TypeError: can't set attributes of built-in/extension type 'list'

>>> list.\_\_dict\_\_['append']

<method 'append' of 'list' objects>

>>> list.\_\_dict\_\_['answer'] = 42

Traceback (most recent call last):

File "<stdin>", line 1, in ?

TypeError: object does not support item assignment

>>> class L(list):

... pass

...

>>> L.append = list.append

>>> L.answer = 42

>>> L.\_\_dict\_\_['answer']

42

>>> L.\_\_dict\_\_['answer'] = 42

Traceback (most recent call last):

File "<stdin>", line 1, in ?

TypeError: object does not support item assignment

>>>

For the curious: there are two reasons why changing built-in classes is disallowed. First, it would be too easy to break an invariant of a built-in type that is relied upon elsewhere, either by the standard library, or by the run-time code. Second, when Python is embedded in another application that creates multiple Python interpreters, the built-in class objects (being statically allocated data structures) are shared between all interpreters; thus, code running in one interpreter might wreak havoc on another interpreter, which is a no-no.

### Static methods and class methods

The new descriptor API makes it possible to add static methods and class methods. Static methods are easy to describe: they behave pretty much like static methods in C++ or Java. Here's an example:

class C:

def foo(x, y):

print "staticmethod", x, y

foo = staticmethod(foo)

C.foo(1, 2)

c = C()

c.foo(1, 2)

Both the call C.foo(1, 2) and the call c.foo(1, 2) call foo() with two arguments, and print "staticmethod 1 2". No "self" is declared in the definition of foo(), and no instance is required in the call. If an instance is used, it is only used to find the class that defines the static method. This works for classic and new classes!

The line "foo = staticmethod(foo)" in the class statement is the crucial element: this makes foo() a static method. The built-in staticmethod() wraps its function argument in a special kind of descriptor whose \_\_get\_\_() method returns the original function unchanged.

More on \_\_get\_\_ methods: in Python 2.2, the magic of binding methods to instances (even for classic classes!) is done through the \_\_get\_\_ method of the object found in the class. The \_\_get\_\_ method for regular function objects returns a bound method object; the \_\_get\_\_ method for staticfunction objects returns the underlying function. If a class attribute has no \_\_get\_\_ method, it is never bound to an instance, or in other words there's a default \_\_get\_\_ operation that returns the object unchanged; this is how simple class variables (for example numerical values) are handled.

Class methods use a similar pattern to declare methods that receive an implicit first argument that is the *class* for which they are invoked. This has no C++ or Java equivalent, and is not quite the same as what class methods are in Smalltalk, but may serve a similar purpose. (Python also has real [metaclasses](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclasses), and perhaps methods defined in a metaclass have more right to the name "class method"; but I expect that most programmers won't be using metaclasses.) Here's an example:

class C:

def foo(cls, y):

print "classmethod", cls, y

foo = classmethod(foo)

C.foo(1)

c = C()

c.foo(1)

Both the call C.foo(1) and the call c.foo(1) end up calling foo() with *two* arguments, and print "classmethod \_\_main\_\_.C 1". The first argument of foo() is implied, and it is the class, even if the method was invoked via an instance. Now let's continue the example:

class D(C):

pass

D.foo(1)

d = D()

d.foo(1)

This prints "classmethod \_\_main\_\_.D 1" both times; in other words, the class passed as the first argument of foo() is the class involved in the call, not the class involved in the definition of foo().

But notice this:

class E(C):

def foo(cls, y): # override C.foo

print "E.foo() called"

C.foo(y)

foo = classmethod(foo)

E.foo(1)

e = E()

e.foo(1)

In this example, the call to C.foo() from E.foo() will see class C as its first argument, not class E. This is to be expected, since the call specifies the class C. But it stresses the difference between these class methods and methods defined in [metaclasses](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclasses), where an upcall to a metamethod would pass the target class as an explicit first argument. (If you don't understand this, don't worry, you're not alone. :-)

### Properties: attributes managed by get/set methods

Properties are a neat way to implement attributes whose *usage* resembles attribute access, but whose *implementation* uses method calls. These are sometimes known as "managed attributes". In prior Python versions, you could only do this by overriding \_\_getattr\_\_ and \_\_setattr\_\_; but overriding \_\_setattr\_\_ slows down *all* attribute assignments considerably, and overriding \_\_getattr\_\_ is always a bit tricky to get right. Properties let you do this painlessly, without having to override \_\_getattr\_\_ or \_\_setattr\_\_.

I'll show an example first. Let's define a class with an attribute x defined by a pair of methods, getx() and setx():

class C(object):

def \_\_init\_\_(self):

self.\_\_x = 0

def getx(self):

return self.\_\_x

def setx(self, x):

if x < 0: x = 0

self.\_\_x = x

x = property(getx, setx)

Here's a small demonstration:

>>> a = C()

>>> a.x = 10

>>> print a.x

10

>>> a.x = -10

>>> print a.x

0

>>> a.setx(12)

>>> print a.getx()

12

>>>

The full signature is property(fget=None, fset=None, fdel=None, doc=None). The fget, fset and fdel arguments are the methods called when the attribute is get, set or deleted. If any of these three is unspecified or None, the corresponding operation will raise an AttributeError exception. The fourth argument is the doc string for the attribute; it can be retrieved from the class as the following example shows:

>>> class C(object):

... def getx(self): return 42

... x = property(getx, doc="hello")

...

>>> C.x.\_\_doc\_\_

'hello'

>>>

Things to notice about property() (all advanced material except the first one):

* **Properties do not work for classic classes**, but you don't get a clear error when you try this. Your get method will be called, so it appears to work, but upon attribute assignment, a classic class instance will simply set the value in its \_\_dict\_\_ without calling the property's set method, and after that, the property's get method won't be called either. (You could override \_\_setattr\_\_ to fix this, but it would be prohibitively expensive.)

As far as property() is concerned, its fget, fset and fdel arguments are functions, not methods - they are passed an explicit reference to the object as their first argument. Since property() is typically used in a class statement, this is correct (the methods really *are* function objects at the time when property() is called) but you can still think of them as methods - as long as you aren't using a [metaclass](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclasses) that does special things to methods.

* The get method won't be called when the property is accessed as a class attribute (C.x) instead of as an instance attribute (C().x). If you want to override the \_\_get\_\_ operation for properties when used as a class attribute, you can subclass property - it is a new-style type itself - to extend its \_\_get\_\_ method, or you can define a descriptor type from scratch by creating a new-style class that defines \_\_get\_\_, \_\_set\_\_ and \_\_delete\_\_ methods.

### Method resolution order

With multiple inheritance comes the question of method resolution order: the order in which a class and its bases are searched looking for a method of a given name.

In classic Python, the rule is given by the following recursive function, also known as the left-to-right depth-first rule:

def classic\_lookup(cls, name):

"Look up name in cls and its base classes."

if cls.\_\_dict\_\_.has\_key(name):

return cls.\_\_dict\_\_[name]

for base in cls.\_\_bases\_\_:

try:

return classic\_lookup(base, name)

except AttributeError:

pass

raise AttributeError, name

In Python 2.2, I've decided to adopt a different lookup rule for new-style classes. (The rule for classic classes remains unchanged for backwards compatibility considerations; eventually all classes will be new-style classes and then the distinction will go away.) I'll try to explain what's wrong with the classic rule first.

The problem with the classic rule becomes apparent when we consider a "diamond diagram":

class A:

^ ^ def save(self): ...

/ \

/ \

/ \

/ \

class B class C:

^ ^ def save(self): ...

\ /

\ /

\ /

\ /

class D

Arrows point from a subtype to its base type(s). This particular diagram means B and C derive from A, and D derives from B and C (and hence also, indirectly, from A).

Assume that C overrides the method save(), which is defined in the base A. (C.save() probably calls A.save() and then saves some of its own state.) B and D don't override save(). When we invoke save() on a D instance, which method is called? According to the classic lookup rule, A.save() is called, ignoring C.save()!

This is not good. It probably breaks C (its state doesn't get saved), defeating the whole purpose of inheriting from C in the first place.

Why wasn't this a problem in classic Python? Diamond diagrams are rarely found in classic Python class hierarchies. Most class hierarchies use single inheritance, and multiple inheritance is usually limited to mix-in classes. In fact, the problem shown here is probably the reason why multiple inheritance is unpopular in classic Python!

Why will this be a problem in the new system? The 'object' type at the top of the type hierarchy defines a number of methods that can usefully be extended by subtypes, for example \_\_getattribute\_\_() and \_\_setattr\_\_().

(Aside: the \_\_getattr\_\_() method is not really the implementation for the get-attribute operation; it is a hook that only gets invoked when an attribute cannot be found by normal means. This has often been cited as a shortcoming - some class designs have a legitimate need for a get-attribute method that gets called for *all* attribute references, and this problem is solved now by making \_\_getattribute\_\_() available. But then this method has to be able to invoke the default implementation somehow. The most natural way is to make the default implementation available as object.\_\_getattribute\_\_(self, name).)

Thus, a classic class hierarchy like this:

class B class C:

^ ^ \_\_setattr\_\_()

\ /

\ /

\ /

\ /

class D

will change into a diamond diagram under the new system:

object:

^ ^ \_\_setattr\_\_()

/ \

/ \

/ \

/ \

class B class C:

^ ^ \_\_setattr\_\_()

\ /

\ /

\ /

\ /

class D

and while in the original diagram C.\_\_setattr\_\_() is invoked, under the new system with the classic lookup rule, object.\_\_setattr\_\_() would be invoked!

Fortunately, there's a lookup rule that's better. It's a bit difficult to explain, but it does the right thing in the diamond diagram, and it is the same as the classic lookup rule when there are no diamonds in the inheritance graph (when it is a tree).

The new lookup rule constructs a list of all classes in the inheritance diagram in the order in which they will be searched. This construction is done when the class is defined, to save time. To explain the new lookup rule, let's first consider what such a list would look like for the classic lookup rule. Note that in the presence of diamonds the classic lookup visits some classes multiple times. For example, in the ABCD diamond diagram above, the classic lookup rule visits the classes in this order:

D, B, A, C, A

Note how A occurs twice in the list. The second occurrence is redundant, since anything that could be found there would already have been found when searching the first occurrence. But it is visited nonetheless (the recursive implementation of the classic rule doesn't remember which classes it has already visited).

We use this observation to explain our new lookup rule. Using the classic lookup rule, construct the list of classes that would be searched, including duplicates. Now for each class that occurs in the list multiple times, remove all occurrences except for the last. The resulting list contains each ancestor class exactly once (including the most derived class, D in the example): D, B, C, A.

Searching for methods in this order will do the right thing for the diamond diagram. Because of the way the list is constructed, it never changes the search order in situations where no diamond is involved.

Isn't this backwards incompatible? Won't it break existing code? It would, if we changed the method resolution order for all classes. However, in Python 2.2, the new lookup rule will only be applied to types derived from built-in types, which is a new feature. Class statements without a base class create "classic classes", and so do class statements whose base classes are themselves classic classes. For classic classes the classic lookup rule will be used. We may also provide a tool that analyzes a class hierarchy looking for methods that would be affected by a change in method resolution order.

#### Order Disagreements

This section is for advanced readers only. The current implementation uses a subtly different algorithm, which may yield a somewhat different search order in rare cases. The difference only shows up when two given base classes occur in a different order in the inheritance list of two different derived classes, and those derived classes are both inherited by yet another class. Here's the smallest example I can think of:

class A(object):

def meth(self): return "A"

class B(object):

def meth(self): return "B"

class X(A, B): pass

class Y(B, A): pass

class Z(X, Y): pass

According to the algorithm given above, Z's MRO (Method Resolution order) should be [Z, X, Y, B, A, object]. But if you try this in Python 2.2 (using Z.\_\_mro\_\_, see [below](http://www.python.org/download/releases/2.2/descrintro/" \l "cooperation)), you get [Z, X, Y, A, B, object]! In a future version, two things might happen: Z's MRO might change to [Z, X, Y, B, A, object]; or the declaration of class Z might become illegal because it introduces a "order disagreement": class A precedes B in X's inheritance list, but follows it in Y's inheritance list.

The book ["Putting Metaclasses to Work"](http://www.python.org/download/releases/2.2/descrintro/" \l "references), which inspired me to change the MRO, defines the MRO algorithm that's currently implemented, but its description of the algorithm is pretty hard to grasp - I didn't even realize that the algorithm above doesn't always compute the same MRO until [Tim Peters](http://www.python.org/tim_one/) found a counterexample. Fortunately, counterexamples can only occur when there are order disagreements in the inheritance graph. The book outlaws classes containing such order disagreements, if the order disagreement is "serious". An order disagreement between two classes is serious when the two classes define at least one method with the same name. In the example above, the order disagreement is serious. In Python 2.2, I chose not to check for serious order disagreements; but the meaning of a program containing a serious order disagreement is undefined, and its effect may change in the future.

### Cooperative methods and "super"

One of the coolest, but perhaps also one of the most unusual features of the new classes is the possibility to write "cooperative" classes. Cooperative classes are written with multiple inheritance in mind, using a pattern that I call a "cooperative super call". This is known in some other multiple-inheritance languages as "call-next-method", and is more powerful than the super call found in single-inheritance languages like Java or Smalltalk. C++ has neither form of super call, relying instead on an explicit mechanism similar to that used in classic Python. (The term "cooperative method" comes from ["Putting Metaclasses to Work"](http://www.python.org/download/releases/2.2/descrintro/" \l "references).)

As a refresher, let's first review the traditional, non-cooperative super call. When a class C derives from a base class B, C often wants to override a method m defined in B. A "super call" occurs when C's definition of m calls B's definition of m to do some of its work. In Java, the body of m in C can write super(a, b, c) to call B's definition of m with argument list (a, b, c). In Python, C.m writes B.m(self, a, b, c) to accomplish the same effect. For example:

class B:

def m(self):

print "B here"

class C(B):

def m(self):

print "C here"

B.m(self)

We say that C's method m "extends" B's method m. The pattern here works well as long as we're using single inheritance, but it breaks down with multiple inheritance. Let's look at four classes whose inheritance diagram forms a "diamond" (the same diagram was shown graphically in the previous section):

class A(object): ..

class B(A): ...

class C(A): ...

class D(B, C): ...

Suppose A defines a method m, which is extended by both B and C. Now what is D to do? It inherits two implementations of m, one from B and one from C. Traditionally, Python simply picks the first one found, in this case the definition from B. This is not ideal, because this completely ignores C's definition. To see what's wrong with ignoring C's m, assume that these classes represent some kind of persistent container hierarchy, and consider a method that implements the operation "save your data to disk". Presumably, a D instance has both B's data and C's data, as well as A's data (a single copy of the latter). Ignoring C's definition of the save method would mean that a D instance, when requested to save itself, only saves the A and B parts of its data, but not the part of its data defined by class C!

C++ notices that D inherits two conflicting definitions of method m, and issues an error message. The author of D is then supposed to override m to resolve the conflict. But what is D's definition of m supposed to do? It can call B's m followed by C's m, but because both definitions call the definition of m inherited from A, A's m ends up being called twice! Depending on the details of the operation, this is at best an inefficiency (when m is idempotent), at worst an error. Classic Python has the same problem, except it doesn't even consider it an error to inherit two conflicting definitions of a method: it simply picks the first one.

The traditional solution to this dilemma is to split each derived definition of m into two parts: a partial implementation \_m, which only saves the data that is unique to one class, and a full implementation m, which calls its own \_m and that of the base class(es). For example:

class A(object):

def m(self): "save A's data"

class B(A):

def \_m(self): "save B's data"

def m(self): self.\_m(); A.m(self)

class C(A):

def \_m(self): "save C's data"

def m(self): self.\_m(); A.m(self)

class D(B, C):

def \_m(self): "save D's data"

def m(self): self.\_m(); B.\_m(self); C.\_m(self); A.m(self)

There are several problems with this pattern. First of all, there is the proliferation of extra methods and calls. But perhaps more importantly, it creates an undesirable dependency in the derived classes on details of the dependency graph of their base classes: the existence of A can no longer be considered an implementation detail of B and C, since class D needs to know about it. If, in a future version of the program, we want to remove the dependency on A from B and C, this will affect derived classes like D as well; likewise, if we want to add another base class AA to B and C, all their derived classes must be updated as well.

The "call-next-method" pattern solves this problem nicely, in combination with the new method resolution order. Here's how:

class A(object):

def m(self): "save A's data"

class B(A):

def m(self): "save B's data"; super(B, self).m()

class C(A):

def m(self): "save C's data"; super(C, self).m()

class D(B, C):

def m(self): "save D's data"; super(D, self).m()

Note that the first argument to super is always the class in which it occurs; the second argument is always self. Also note that self is not repeated in the argument list for m.

Now, in order to explain how super works, consider the MRO for each of these classes. The MRO is given by the \_\_mro\_\_ class attribute:

A.\_\_mro\_\_ == (A, object)

B.\_\_mro\_\_ == (B, A, object)

C.\_\_mro\_\_ == (C, A, object)

D.\_\_mro\_\_ == (D, B, C, A, object)

The expression super(C, self).m should only be used inside the implementation of method m in class C. Bear in mind that while self is an instance of C, self.\_\_class\_\_ may not be C: it may be a class derived from C (for example, D). The expression super(C, self).m, then, searches self.\_\_class\_\_.\_\_mro\_\_ (the MRO of the class that was used to create the instance in self) for the occurrence of C, and then starts looking for an implementation of method m *following* that point.

For example, if self is a C instance, super(C, self).m will find A's implementation of m, as will super(B, self).m if self is a B instance. But now consider a D instance. In D's m, super(D, self).m() will find and call B.m(self), since B is the first base class following D in D.\_\_mro\_\_ that defines m. Now in B.m, super(B, self).m() is called. Since self is a D instance, the MRO is (D, B, C, A, object) and the class following B is C. This is where the search for a definition of m continues. This finds C.m, which is called, and in turn calls super(C, self).m(). Still using the same MRO, we see that the class following C is A, and thus A.m is called. This is the original definition of m, so no super call is made at this point.

Note how the same super expression finds a different class implementing a method depending on the class of self! This is the crux of the cooperative super mechanism.

The super call as shown above is somewhat prone to errors: it is easy to copy and paste a super call from one class to another while forgetting to change the class name to that of the target class, and this mistake won't be detected if both classes are part of the same inheritance graph. (You can even cause infinite recursion by mistakenly passing in the name of a derived class of the class containing the super call.) It would be nice if we didn't have to name the class explicitly, but this would require more help from Python's parser than we can currently get. I hope to fix this in a future Python release by making the parser recognize super.

In the mean time, here's a trick you can apply. We can create a class variable named \_\_super that has "binding" behavior. (Binding behavior is a new concept in Python 2.2, but it formalizes a well-known concept from classic Python: the transformation from an unbound method to a bound method when it is accessed via the getattr operation on an instance. It is implemented by the \_\_get\_\_ method discussed [above](http://www.python.org/download/releases/2.2/descrintro/" \l "staticmethods).) Here's a simple example:

class A:

def m(self): "save A's data"

class B(A):

def m(self): "save B's data"; self.\_\_super.m()

B.\_B\_\_super = super(B)

class C(A):

def m(self): "save C's data"; self.\_\_super.m()

C.\_C\_\_super = super(C)

class D(B, C):

def m(self): "save D's data"; self.\_\_super.m()

D.\_D\_\_super = super(D)

Part of the trick is in the use of the name \_\_super, which (through the name mangling transformation) contains the class name. This ensures that self.\_\_super means something different in each class (as long as the class names differ; unfortunately, it *is* possible in Python to reuse the name of a base class for a derived class). Another part of the trick is that the super built-in can be called with a single argument, and then creates an unbound version that can be bound by a later instance getattr operation.

Unfortunately, this example is still rather ugly for a number of reasons: super requires that the class is passed in, but the class is not available until after execution of the class statement is completed, so the \_\_super class attribute must be assigned outside the class. Outside the class, name mangling doesn't work (after all it is intended to be a privacy feature) so the assignment must use the unmangled name. Fortunately, it's possible to write a [metaclass](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclasses) that automatically adds a \_\_super attribute to its classes; see the [autosuper metaclass example below](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclass_examples).

Note that super(class, subclass) also works; this is needed for [\_\_new\_\_](http://www.python.org/download/releases/2.2/descrintro/" \l "__new__) and other static methods.

#### Example: coding super in Python.

As an illustration of the power of the new system, here's a fully functional implementation of the super() built-in class in pure Python. This may also help clarify the semantics of super() by spelling out the search in ample detail. The print statement at the bottom of the following code prints "DCBA".

class Super(object):

def \_\_init\_\_(self, type, obj=None):

self.\_\_type\_\_ = type

self.\_\_obj\_\_ = obj

def \_\_get\_\_(self, obj, type=None):

if self.\_\_obj\_\_ is None and obj is not None:

return Super(self.\_\_type\_\_, obj)

else:

return self

def \_\_getattr\_\_(self, attr):

if isinstance(self.\_\_obj\_\_, self.\_\_type\_\_):

starttype = self.\_\_obj\_\_.\_\_class\_\_

else:

starttype = self.\_\_obj\_\_

mro = iter(starttype.\_\_mro\_\_)

for cls in mro:

if cls is self.\_\_type\_\_:

break

# Note: mro is an iterator, so the second loop

# picks up where the first one left off!

for cls in mro:

if attr in cls.\_\_dict\_\_:

x = cls.\_\_dict\_\_[attr]

if hasattr(x, "\_\_get\_\_"):

x = x.\_\_get\_\_(self.\_\_obj\_\_)

return x

raise AttributeError, attr

class A(object):

def m(self):

return "A"

class B(A):

def m(self):

return "B" + Super(B, self).m()

class C(A):

def m(self):

return "C" + Super(C, self).m()

class D(C, B):

def m(self):

return "D" + Super(D, self).m()

print D().m() # "DCBA"

### Overriding the \_\_new\_\_ method

When subclassing immutable built-in types like numbers and strings, and occasionally in other situations, the static method \_\_new\_\_ comes in handy. \_\_new\_\_ is the first step in instance construction, invoked *before* \_\_init\_\_. The \_\_new\_\_ method is called with the class as its first argument; its responsibility is to return a new instance of that class. Compare this to \_\_init\_\_: \_\_init\_\_ is called with an instance as its first argument, and it doesn't return anything; its responsibility is to initialize the instance. There are situations where a new instance is created without calling \_\_init\_\_ (for example when the instance is loaded from a pickle). There is no way to create a new instance without calling \_\_new\_\_ (although in some cases you can get away with calling a base class's \_\_new\_\_).

Recall that you create class instances by calling the class. When the class is a new-style class, the following happens when it is called. First, the class's \_\_new\_\_ method is called, passing the class itself as first argument, followed by any (positional as well as keyword) arguments received by the original call. This returns a new instance. Then that instance's \_\_init\_\_ method is called to further initialize it. (This is all controlled by the \_\_call\_\_ method of the metaclass, by the way.)

Here is an example of a subclass that overrides \_\_new\_\_ - this is how you would normally use it.

>>> class inch(float):

... "Convert from inch to meter"

... def \_\_new\_\_(cls, arg=0.0):

... return float.\_\_new\_\_(cls, arg\*0.0254)

...

>>> print inch(12)

0.3048

>>>

This class isn't very useful (it's not even the right way to go about unit conversions) but it shows how to extend the constructor of an immutable type. If instead of \_\_new\_\_ we had tried to override \_\_init\_\_, it wouldn't have worked:

>>> class inch(float):

... "THIS DOESN'T WORK!!!"

... def \_\_init\_\_(self, arg=0.0):

... float.\_\_init\_\_(self, arg\*0.0254)

...

>>> print inch(12)

12.0

>>>

The version overriding \_\_init\_\_ doesn't work because the float type's \_\_init\_\_ is a no-op: it returns immediately, ignoring its arguments.

All this is done so that immutable types can preserve their immutability while allowing subclassing. If the value of a float object were initialized by its \_\_init\_\_ method, you could change the value of an existing float object! For example, this would work:

>>> # THIS DOESN'T WORK!!!

>>> import math

>>> math.pi.\_\_init\_\_(3.0)

>>> print math.pi

3.0

>>>

I could have fixed this problem in other ways, for example by adding an "already initialized" flag or only allowing \_\_init\_\_ to be called on subclass instances, but those solutions are inelegant. Instead, I added \_\_new\_\_, which is a perfectly general mechanism that can be used by built-in and user-defined classes, for immutable and mutable objects.

Here are some rules for \_\_new\_\_:

* \_\_new\_\_ is a static method. When defining it, you don't need to (but may!) use the phrase "\_\_new\_\_ = staticmethod(\_\_new\_\_)", because this is implied by its name (it is special-cased by the class constructor).
* The first argument to \_\_new\_\_ must be a class; the remaining arguments are the arguments as seen by the constructor call.
* A \_\_new\_\_ method that overrides a base class's \_\_new\_\_ method may call that base class's \_\_new\_\_ method. The first argument to the base class's \_\_new\_\_ method call should be the class argument to the overriding \_\_new\_\_ method, not the base class; if you were to pass in the base class, you would get an instance of the base class.
* Unless you want to play games like those described in the next two bullets, a \_\_new\_\_ method *must* call its base class's \_\_new\_\_ method; that's the only way to create an instance of your object. The subclass \_\_new\_\_ can do two things to affect the resulting object: pass different arguments to the base class \_\_new\_\_, and modify the resulting object after it's been created (for example to initialize essential instance variables).
* \_\_new\_\_ must return an object. There's nothing that requires that it return a new object that is an instance of its class argument, although that is the convention. If you return an existing object, the constructor call will still call its \_\_init\_\_ method. If you return an object of a different class, *its* \_\_init\_\_ method will be called. If you forget to return something, Python will unhelpfully return None, and your caller will probably be very confused.
* For immutable classes, your \_\_new\_\_ may return a cached reference to an existing object with the same value; this is what the int, str and tuple types do for small values. This is one of the reasons why their \_\_init\_\_ does nothing: cached objects would be re-initialized over and over. (The other reason is that there's nothing left for \_\_init\_\_ to initialize: \_\_new\_\_ returns a fully initialized object.)
* If you subclass a built-in immutable type and want to add some mutable state (maybe you add a default conversion to a string type), it's best to initialize the mutable state in the \_\_init\_\_ method and leave \_\_new\_\_ alone.
* If you want to change the constructor's signature, you often have to override both \_\_new\_\_ and \_\_init\_\_ to accept the new signature. However, most built-in types ignore the arguments to the method they don't use; in particular, the immutable types (int, long, float, complex, str, unicode, and tuple) have a dummy \_\_init\_\_, while the mutable types (dict, list, file, and also super, classmethod, staticmethod, and property) have a dummy \_\_new\_\_. The built-in type 'object' has a dummy \_\_new\_\_ and a dummy \_\_init\_\_ (which the others inherit). The built-in type 'type' is special in many respects; see the section on [metaclasses](http://www.python.org/download/releases/2.2/descrintro/" \l "metaclasses).
* (This has nothing to do to \_\_new\_\_, but is handy to know anyway.) If you subclass a built-in type, extra space is automatically added to the instances to accomodate \_\_dict\_\_ and \_\_weakrefs\_\_. (The \_\_dict\_\_ is not initialized until you use it though, so you shouldn't worry about the space occupied by an empty dictionary for each instance you create.) If you don't need this extra space, you can add the phrase "\_\_slots\_\_ = []" to your class. (See [above](http://www.python.org/download/releases/2.2/descrintro/" \l "subclassing) for more about \_\_slots\_\_.)
* Factoid: \_\_new\_\_ is a static method, not a class method. I initially thought it would have to be a class method, and that's why I added the classmethod primitive. Unfortunately, with class methods, upcalls don't work right in this case, so I had to make it a static method with an explicit class as its first argument. Ironically, there are now no known uses for class methods in the Python distribution (other than in the test suite). I might even get rid of classmethod in a future release if no good use for it can be found!

As another example of \_\_new\_\_, here's a way to implement the singleton pattern.

class Singleton(object):

def \_\_new\_\_(cls, \*args, \*\*kwds):

it = cls.\_\_dict\_\_.get("\_\_it\_\_")

if it is not None:

return it

cls.\_\_it\_\_ = it = object.\_\_new\_\_(cls)

it.init(\*args, \*\*kwds)

return it

def init(self, \*args, \*\*kwds):

pass

To create a singleton class, you subclass from Singleton; each subclass will have a single instance, no matter how many times its constructor is called. To further initialize the subclass instance, subclasses should override 'init' instead of \_\_init\_\_ - the \_\_init\_\_ method is called each time the constructor is called. For example:

>>> class MySingleton(Singleton):

... def init(self):

... print "calling init"

... def \_\_init\_\_(self):

... print "calling \_\_init\_\_"

...

>>> x = MySingleton()

calling init

calling \_\_init\_\_

>>> assert x.\_\_class\_\_ is MySingleton

>>> y = MySingleton()

calling \_\_init\_\_

>>> assert x is y

>>>

### Metaclasses

In the past, the subject of metaclasses in Python has caused hairs to raise and even brains to explode (see, for example [Metaclasses in Python 1.5](http://www.python.org/download/releases/2.2/descrintro/" \l "references)). Fortunately, in Python 2.2, metaclasses are more accessible and less dangerous.

Terminology-wise, a metaclass is simply "the class of a class". Any class whose instances are themselves classes, is a metaclass. When we talk about an instance that's not a class, the instance's metaclass is the class of its class: by definition, x's metaclass is x.\_\_class\_\_.\_\_class\_\_. But when we talk about a class C, we often refer to its metaclass when we mean C.\_\_class\_\_ (not C.\_\_class\_\_.\_\_class\_\_, which would be a meta-metaclass; there's not much use for those although we don't rule them out).

The built-in 'type' is the most common metaclass; it is the metaclass of all built-in types. Classic classes use a different metaclass: the type known as types.ClassType. The latter is relatively uninteresting; it's a historical artefact that's needed to give classic classes their classic behavior. You can't get to the metaclass of a classic instance using x.\_\_class\_\_.\_\_class\_\_; you have to use type(x.\_\_class\_\_), because classic classes don't support the \_\_class\_\_ attribute on classes (only on instances).

When a class statement is executed, the interpreter first determines the appropriate metaclass M, and then calls M(name, bases, dict). All this happens at the *end* of the class statement, after the body of the class (where methods and class variables are defined) has already been executed. The arguments to M are the class name (a string taken from the class statement), a tuple of base classes (expressions evaluated at the start of the class statement; this is () if no bases are specified in the class statement), and a dictionary containing the methods and class variables defined by the class statement. Whatever this call M(name, bases, dict) returns is then assigned to the variable corresponding to the class name, and that's all there is to the class statement.

How is M determined?

* If dict['\_\_metaclass\_\_'] exists, it is used.
* Otherwise, if there is at least one base class, its metaclass is used (this looks for a \_\_class\_\_ attribute first and if that's not found, uses its type). (In classic Python, this step existed too, but was only executed when the metaclass was callable. This was called the Don Beaudry hook - may it rest in peace.)
* Otherwise, if there's a global variable named \_\_metaclass\_\_, it is used.
* Otherwise, the classic metaclass (types.ClassType) is used.

The most common outcomes here are that M is either types.ClassType (creating a classic class), or 'type' (creating a new-style class). Other common outcomes are a custom extension type (like Jim Fulton's ExtensionClass), or a subtype of 'type' (when we're using new-style metaclasses). But it's possible to have something completely outlandish here: if we specify a base class that has a custom \_\_class\_\_ attribute, we can use anything as a "metaclass". That was the brain-exploding topic of my original [metaclass paper](http://www.python.org/download/releases/2.2/descrintro/" \l "references), and I won't repeat it here.

There's always an additional wrinkle. When you mix classic classes and new-style classes in the list of bases, the metaclass of the first new-style base class is used instead of types.ClassType (assuming dict['\_\_metaclass\_\_'] is undefined). The effect is that when you cross a classic class and a new-style class, the offspring is a new-style class.

And another one (I promise this is the last wrinkle in the metaclass determination). For new-style metaclasses, there is a constraint that the chosen metaclass is equal to, or a subclass of, each of the metaclasses of the bases. Consider a class C with two base classes, B1 and B2. Let's say M = C.\_\_class\_\_, M1 = B1.\_\_class\_\_, M2 = B2.\_\_class\_\_. Then we require issubclass(M, M1) and issubclass(M, M2). (This is because a method of B1 should be able to call a meta-method defined in M1 on self.\_\_class\_\_, even when self is an instance of a subclass of B1.)

The [metaclasses book](http://www.python.org/download/releases/2.2/descrintro/" \l "references) describes a mechanism whereby a suitable metaclass is automatically created, when necessary, through multiple inheritance from M1 and M2. In Python 2.2, I have chosen a simpler approach which raises an exception if the metaclass constraint is not satisfied; it is up to the programmer to provide a suitable metaclass through the \_\_metaclass\_\_ class variable. However, if one of the base metaclasses satisfies the constraint (including the explicitly given \_\_metaclass\_\_, if any), the first base metaclass found satisfying the constraint will be used as the metaclass.

In practice, this means that if you have a degenerate metaclass hierarchy that has the shape of a tower (meaning that for two metaclasses M1 and M2, at least one of issubclass(M1, M2) or issubclass(M2, M1) is always true), you don't have to worry about the metaclass constraint. For example:

# Metaclasses

class M1(type): ...

class M2(M1): ...

class M3(M2): ...

class M4(type): ...

# Regular classes

class C1:

\_\_metaclass\_\_ = M1

class C2(C1):

\_\_metaclass\_\_ = M2

class C3(C1, C2):

\_\_metaclass\_\_ = M3

class D(C2, C3):

\_\_metaclass\_\_ = M1

class C4:

\_\_metaclass\_\_ = M4

class E(C3, C4):

pass

For class C2, the constraint is satisfied because M2 is a subclass of M1. For class C3, it is satisfied because M3 is a subclass of both M1 and M2. For class D, the explicit metaclass M1 is not a subclass of the base metaclasses (M2, M3), but choosing M3 satisfies the constraint, so D.\_\_class\_\_ is M3. However, class E is an error: the two metaclasses involved are M3 and M4, and neither is a subclass of the other. We can fix this latter case as follows:

# A new metaclass

class M5(M3, M4): pass

# Fixed class E

class E(C3, C4):

\_\_metaclass\_\_ = M5

(The approach from the metaclasses book would automatically supply the class definition for M5 given the original definition of class E.)

#### Metaclass examples

Let's refresh some theory first. Remember that a class statement causes a call to M(name, bases, dict) where M is the metaclass. Now, a metaclass is a class, and we've already established that when a class is called, its \_\_new\_\_ and \_\_init\_\_ methods are called in sequence. Therefore, something like this will happen:

cls = M.\_\_new\_\_(M, name, bases, dict)

assert cls.\_\_class\_\_ is M

M.\_\_init\_\_(cls, name, bases, dict)

I'm writing the \_\_init\_\_ call as an unbound method call here. This clarifies that we're calling the \_\_init\_\_ defined by M, not the \_\_init\_\_ defined in cls (which would be the initialization for instances of cls). But it really calls the \_\_init\_\_ method of object cls; cls just happens to be a class.

Our first example is a metaclass that looks through the methods of a class for methods named \_get\_<something> and \_set\_<something>, and automatically adds property descriptors named <something>. It turns out that it's sufficient to override \_\_init\_\_ to do what we want. The algorithm makes two passes: first it collects names of properties, then it adds them to the class. The collection pass looks through dict, which is the dictionary representing the class variables and methods (excluding base class variables and methods). But the second pass, the property construction pass, looks up \_get\_<something> and \_set\_<something> as class attributes. This means that if a base class defines \_get\_x and a subclass defines \_set\_x, the subclass will have a property x created from both methods, even though only \_set\_x occurs in the subclass's dictionary. Thus, you can extend properties in a subclass. Note that we use the three-argument form of getattr(), so a missing \_get\_x or \_set\_x will be translated into None, not raise an AttributeError. We also call the base class \_\_init\_\_ method, in cooperative fashion using super().

class autoprop(type):

def \_\_init\_\_(cls, name, bases, dict):

super(autoprop, cls).\_\_init\_\_(name, bases, dict)

props = {}

for name in dict.keys():

if name.startswith("\_get\_") or name.startswith("\_set\_"):

props[name[5:]] = 1

for name in props.keys():

fget = getattr(cls, "\_get\_%s" % name, None)

fset = getattr(cls, "\_set\_%s" % name, None)

setattr(cls, name, property(fget, fset))

Let's test autoprop with a silly example. Here's a class that stores an attribute x as its inverted value under self.\_\_x:

class InvertedX:

\_\_metaclass\_\_ = autoprop

def \_get\_x(self):

return -self.\_\_x

def \_set\_x(self, x):

self.\_\_x = -x

a = InvertedX()

assert not hasattr(a, "x")

a.x = 12

assert a.x == 12

assert a.\_InvertedX\_\_x == -12

Our second example creates a class, 'autosuper', which will add a private class variable named \_\_super, set to the value super(cls). (Recall the discussion of self.\_\_super [above](http://www.python.org/download/releases/2.2/descrintro/" \l "cooperation).) Now, \_\_super is a private name (starts with double underscore) but we want it to be a private name of the class to be created, not a private name of autosuper. Thus, we must do the name mangling ourselves, and use setattr() to set the class variable. For the purpose of this example, I'm simplifying the name mangling to "prepend an underscore and the class name". Again, it's sufficient to override \_\_init\_\_ to do what we want, and again, we call the base class \_\_init\_\_ cooperatively.

class autosuper(type):

def \_\_init\_\_(cls, name, bases, dict):

super(autosuper, cls).\_\_init\_\_(name, bases, dict)

setattr(cls, "\_%s\_\_super" % name, super(cls))

Now let's test autosuper with the classic diamond diagram:

class A:

\_\_metaclass\_\_ = autosuper

def meth(self):

return "A"

class B(A):

def meth(self):

return "B" + self.\_\_super.meth()

class C(A):

def meth(self):

return "C" + self.\_\_super.meth()

class D(C, B):

def meth(self):

return "D" + self.\_\_super.meth()

assert D().meth() == "DCBA"

(Our autosuper metaclass is easily fooled if you define a subclass with the same name as a base class; it should really check for that condition and raise an error if it occurs. But that's more code than feels right for an example, so I'll leave it as an exercise for the reader.)

Now we have two independently developed metaclasses, we can combine the two into a third metaclass that inherits from them both:

class autosuprop(autosuper, autoprop):

pass

Simple eh? Because we wrote both metaclasses cooperatively (meaning their methods use super() to call the base class method), that's all we need. Let's test it:

class A:

\_\_metaclass\_\_ = autosuprop

def \_get\_x(self):

return "A"

class B(A):

def \_get\_x(self):

return "B" + self.\_\_super.\_get\_x()

class C(A):

def \_get\_x(self):

return "C" + self.\_\_super.\_get\_x()

class D(C, B):

def \_get\_x(self):

return "D" + self.\_\_super.\_get\_x()

assert D().x == "DCBA"

That's all for today. I hope your brain doesn't hurt too much!

### Backwards incompatibilities

**Relax!** Most features described above are only invoked when you use a class statement with a built-in object as a base class (or when you use an explicit \_\_metaclass\_\_ assignment).

Some things that might affect old code:

* See also the [bugs in 2.2 list](http://www.python.org/download/releases/2.2/descrintro/bugs).
* Introspection works differently (see [PEP 252](http://www.python.org/download/releases/2.2/descrintro/" \l "references)). In particular, most objects now have a \_\_class\_\_ attribute, and the \_\_methods\_\_ and \_\_members\_\_ attributes no longer work, and the dir() function works differently. See also [above](http://www.python.org/download/releases/2.2/descrintro/" \l "introspection).
* Several built-ins that can be seen as coercions or constructors are now type objects rather than factory functions; the type objects support the same behaviors as the old factory functions. Affected are: complex, float, long, int, str, tuple, list, unicode, and type. (There are also new ones: dict, object, classmethod, staticmethod, but since these are new built-ins I can't see how this would break old code.) See also [above](http://www.python.org/download/releases/2.2/descrintro/" \l "factories).
* There's one very specific (and fortunately uncommon) bug that used to go undetected, but which is now reported as an error:
* class A:
* def foo(self): pass
* class B(A): pass
* class C(A):
* def foo(self):
* B.foo(self)

Here, C.foo wants to call A.foo, but by mistake calls B.foo. In the old system, because B doesn't define foo, B.foo is identical to A.foo, so the call would succeed. In the new system, B.foo is marked as a method requiring a B instance, and a C is not a B, so the call fails.

* Binary compatibility with old extensions is not guaranteed. We've tightened this during the alpha and beta release cycle for Python 2.2. As of 2.2b1, Jim Fulton's ExtensionClass works (as shown by a test of Zope 2.4), and I expect that other extensions based on the Don Beaudry hook will work as well. While the ultimate goal of [PEP 253](http://www.python.org/download/releases/2.2/descrintro/" \l "references) is to do away with ExtensionClass, I believe that ExtensionClass should still work in Python 2.2, breaking it no earlier than Python 2.3.

### Additional Topics

These topics should also be discussed:

* descriptors: \_\_get\_\_, \_\_set\_\_, \_\_delete\_\_
* The specs of the built-in types that are subclassable
* The 'object' type and its methods
* <type 'foo'> vs. <type 'mod.foo'> vs. <class 'mod.foo'>