An exploration of post Python 2.1 language features

What we'll try to cover this evening...

- Background on Type:Class Split
- New-style classes (creating, features, inheritance order)
- Descriptors (functions, classmethod, staticmethod, properties, VRML97 fields, BasicProperty)
- Subclassing built-in types (list, str)

Background: The Type/Class Split

- CPython implementation limitation
- Class instances had type InstanceType
  - Only things that didn't were built-ins
  - You couldn't subclass a built-in
- Code had to check for type and class
  - Generic coding became messy guarding against built-in types

Why fix it?

- One of the biggest Python 1.5.2 “Warts”
- Chance to rationalise the class system
  - Generalise mechanisms for customisation
  - Allow for neat features we'll discuss soon...
- It's just cleaner

Accessing new-style classes...

We won't get into metaclasses too deeply...

Don't try this in Python 2.1 or below...

I promise.

Creating/defining new-style classes

- Subclass a new-style class
  - Easiest and the most common approach
  - Works even if there are old-style classes in the inheritance hierarchy
- Built-in “object” is root of the hierarchy
  - Most built-ins are sub-classes of object
- Almost everything is an object sub-class
  - This becomes important eventually...

Ooh, look, new-style classes...

```
class Old:
    pass

class First( object ):
    """Our first new-style class"""
class Second( Old, object ):
    """New-style even with old-style ancestor"""

assert not isinstance( Old, type )
assert isinstance( First, type )
assert isinstance( Second, type )

class Coolstr( str ):
    """Simplistic sub-class of a string"""
class Coollist( list ):
    """Simplistic sub-class of a list"""
assert isinstance( Coolstr('this'), str )
assert isinstance( Coollist( (1,2,3) ), list)
assert isinstance( Coollist( (1,2,3) ), object)
```

Creating/defining new-style classes (cont)

- Metaclass with no old-style classes
  - Useful for defining large number of classes
  - Or converting existing libraries
  - Or when you're using metaclasses anyway

```
__metaclass__ = type
class WithMeta:
    """A New-style class by virtue of metaclass"""

WithMeta2 = type( 'WithMeta2', (), {} )

assert isinstance( WithMeta, type )
assert isinstance( WithMeta2, type )
```

Anatomy of a new-style instance...

- Lots of new __*__ hooks for instances:
  - __new__ - creates instance, as distinct from __init__ which initialises it
  - __getattribute__ - intercept all attribute access
  - __iter__ - explicit iteration protocol
  - __unicode__ - __str__ but for unicode
  - __slots__ - define internal storage

New features on new-style classes themselves...

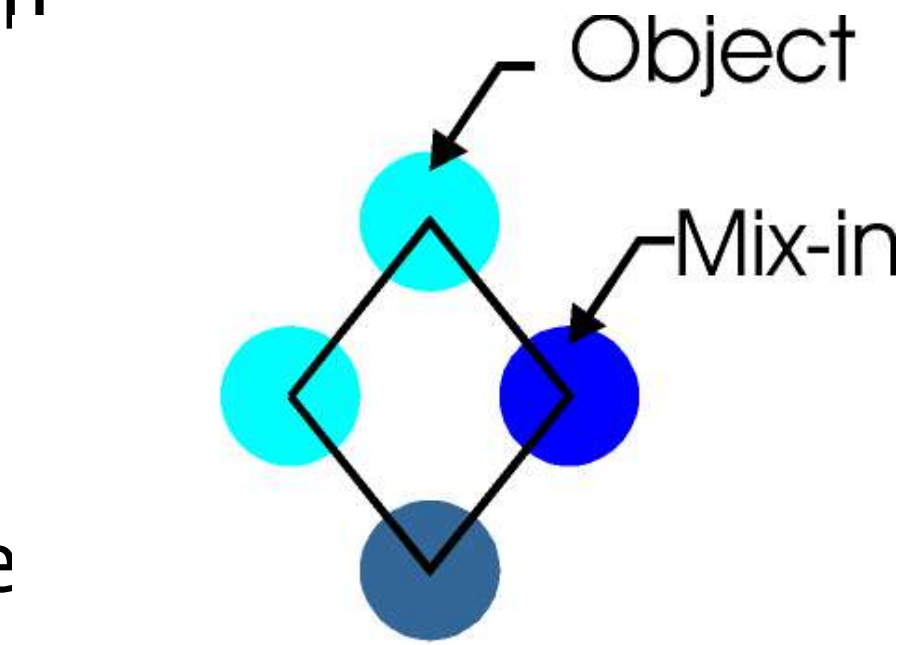
- cls.__class__ -- pointer to the metaclass
  - Normally the built-in “type”
  - Though you'll see others in larger packages
- cls.mro(), cls.__mro__ -- method-resolution order
  - Flattened inheritance hierarchy
  - Only available on the class, not on instances
    - cls.mro() is a meta-method (fairly rare creatures)

New-style classes and inheritance...

- About cls.__mro__ (cls.mro())
  - Eliminates need for recursive lookup
  - __bases__ is still available
- Use a new inheritance ordering
  - Necessary with use of object as base
  - C3 algorithm for 2.3
  - 2.2 used simpler, less desirable algorithm

Why a new inheritance pattern?

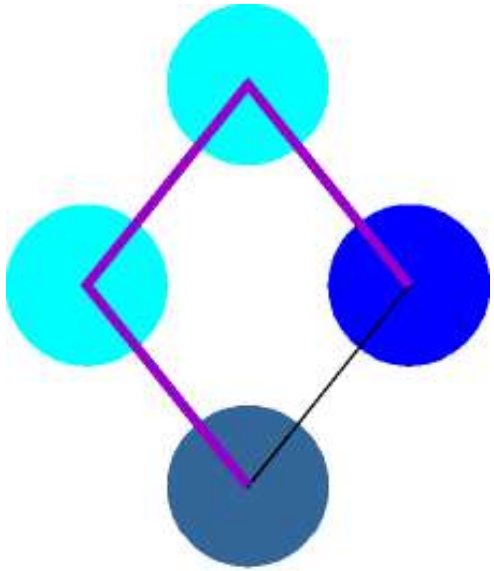
- Multiple inheritance does bad things with "diamond patterns"
- Having everything inherit from object creates lots of these patterns



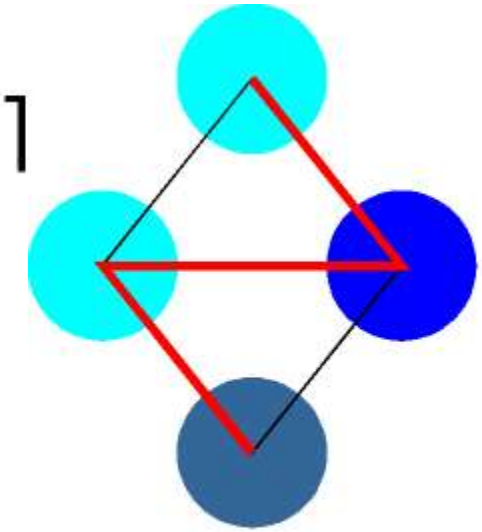
Diamond Inheritance

Dealing with diamonds in 2.1 and 2.3

- Note how mix-in is overridden by object in 2.1
- 2.3's mix-in overrides object as expected



Python 2.1



Python 2.3

Implications of new method-resolution order

- Inheritance tends to be more “natural”
  - Particularly with mix-ins
  - Or complex hierarchies
  - And particularly with diamond inheritance
  - It takes a bit of work to describe C3
- __mro__ helps w/ generic programming
- “super” falls out of the mix quite readily

Cooperative Multiple Inheritance

What's this “super” you're talking about?

Cooperative Multiple Inheritance

- Generic programming ideal
  - Seen commonly in other MI-OO languages
  - Typified by “mix-in classes”
- Mix-in shouldn't need to know which superclass to call
  - That is, it should be generic
  - Want to say “call the appropriate superclass”
- Was a Python 1.5.2 wart w/workarounds

Cooperative Multiple Inheritance and super

- New built-in for Python 2.2+ “super”
- Uses __mro__ of new-style classes
  - Passed-in class is marker for starting search
- Produces a proxy object
- Still has some warts
  - For instance, passing in the marker
    - See autosuper (and variations thereof)
  - Will see others when we subclass built-ins

Cooperative mix-ins for fun and profit

```
__metaclass__ = type
class Base1:
    def x( self, a, b ):
        return a,b
class Base2:
    def x( self, a, b ):
        return a*b, b

class MixInAdder:
    def x( self, a, b ):
        return super(MixInAdder,self).x( a+1, b )

class Final1( MixInAdder, Base1 ):
    pass
class Final2( MixInAdder, Base2 ):
    pass

print Final1().x( 2,3 )
print Final2().x( 2,3 )
```

```
V:\presentations\newstyleclasses>supermixins.py
(3, 3)
(9, 3)
```

Super proxy attribute and method access...

```
__metaclass__ = type
class Y:
    y = 3
    def r( self ):
        pass
class X(Y):
    pass
x = super(X,X())
print x.y
print x.r
y = super(X,X)
print y.y
print y.r
```

```
V:\presentations\newstyleclasses>simplesuper.py
3
<bound method X.r of <__main__.X object at 0x007C9D50>>
3
<unbound method X.r>
```

Descriptors

You've heard about them for years
and most likely never realised
you were using them
every day.

Descriptors, what are they

- Hook for accessing attributes
  - Of an instance or a class
- Descriptor objects are stored in the class
  - Interpreter does descriptor lookup to get/set/delete attribute values
  - We'll ignore meta-descriptors for today, but they work (with some caveats)
- Don't work on old-style classes

Descriptors, what are they (cont)

- "Fall out" of the rationalisation of classes
  - Functions became descriptors that produce:
    - InstanceMethods
    - Unbound InstanceMethods
  - Same thing done before (under the covers), but now can be overridden and/or replaced
    - Just provided a hook and generalised rule
  - Features enabled by the hook are numerous
    - Even in 2.4 era, we're just starting to see the effects of this generalisation being felt

Anatomy of a descriptor

```
class Descriptor( object ):
    """A simple data descriptor"""

    def __get__( self, client, cls ):
        """Retrieve/calculate the value for client instance"""
        print '__get__', client, cls
        if client is not None:
            return client.__dict__[ 'hello' ]
        else:
            return self

    def __set__( self, client, value ):
        """Set the value on the client instance"""
        print '__set__', client, value
        client.__dict__[ 'hello' ] = value

    def __delete__( self, client ):
        """Delete the value from the client instance"""
        print '__delete__', client
        del client.__dict__[ 'hello' ]
```

How the descriptor is called...

- Defined in the new generic get attribute
  - object.__getattribute__ or type.__getattribute__
  - Called for every lookup, so written in C
- object and type different ways of calling
  - desc.__get__(self, cls) vs desc.__get__(None, cls)
- Can override for your own classes
- Diff for data and non-data descriptors...

How the descriptor is called...

```
class object(object):
    def __getattribute__( self, attr ):
        """Pseudo-code for getattribute in object"""
        cls = type(self)
        if cls.__dict__.has_key( attr ):
            desc = cls.__dict__[attr]
            if hasattr( desc, '__get__'):
                # is a actual descriptor...
                if not( hasattr( desc, '__set__') or hasattr
( desc, '__delete__' )):
                    # non-data-descriptor, can be overridden
                    if self.__dict__.has_key( attr ):
                        return self.__dict__[attr]
                return desc.__get__( self, cls )
            else:
                if self.__dict__.has_key( attr ):
                    return self.__dict__[attr]
                else:
                    return desc
        elif self.__dict__.has_key(attr):
            return self.__dict__[attr]
        else:
            raise AttributeError( attr )
```

How the descriptor is called...

```
import descriptor
class X(object):
    s = descriptor.Descriptor()

X.s # access the class' attribute, gets the descriptor
x = X()
x.s = 'this' # set an instance attribute
x.s # retrieve an instance attribute
del x.s # delete an instance attribute
```

```
V:\presentations\newstyleclasses>test_descriptor.py
__get__ None <class '__main__.X'>
__set__ <__main__.X object at 0x007C9CD0> this
__get__ <__main__.X object at 0x007C9CD0> <class
'__main__.X'>
__delete__ <__main__.X object at 0x007C9CD0>
```

Functions as descriptors

- Functions produce (depending on whether accessed via instance or class):
  - Bound instance methods
  - Unbound methods
- “Non-data” descriptors
  - Only __get__, no __set__ or __delete__
  - Difference being that they can be overridden by instance __dict__ values

Functions as descriptors (cont)

```
import types

class Function( types.FunctionType ):
    """Mockup of what a function descriptor looks like"""
    def __get__( self, client, cls ):
        """Retrieve/calculate the value for client instance"""
        if client is not None:
            return types.MethodType( self, client, cls )
        else:
            return types.UnboundMethodType(self, None, cls)
```

Function decorators circa Python 2.2

- Wrap a function to alter operation
- 2.2 introduced two built-in “decorators”
  - classmethod – first argument is always class
  - staticmethod – no extra first argument at all
- These wrappers are extremely common
  - Despite awkward construction
- Lots of calls for a support syntax
  - Huge debate on Python List about final form

About classmethods

- Wrapper around a function
  - First argument is always the class
  - Can be overridden by subclasses
- Common uses:
  - Utility/shared functions for all instances
    - Available on object, so reduces dependencies
    - Also can be called with no instance at all
  - Often used to alter class-local data storage
  - Often used as alternate constructors

How classmethod works (approximately)

```
import types

class classmethod( object ):
    def __init__( self, function ):
        self.function = function
    def __get__( self, client, cls ):
        """Get classmethod for class or instance"""
        if cls is None:
            cls = type(client)
        return types.MethodType(
            self.function,
            cls,
            cls.__class__
        )
```

How classmethod is used

```
class A( object ):
    counter = 0
    def newID( cls ):
        cls.counter += 1
        return '%08x'%(cls.counter,)
    newID = classmethod( newID )

class B( A ):
    def newID( cls ):
        A.counter += 1
        return 'B..%08x'%(cls.counter,)
    newID = classmethod( newID )

items = [A(), B(), A(), B()]
print A.newID()
for item in items:
    print item.newID()
```

```
V:\presentations\newstyleclasses>classmethodsample.py
00000001
00000002
B..00000003
00000004
B..00000005
```

So, on to staticmethod...

- Logical extension of classmethod
  - Instead of method that gets the class as the default argument, a method that gets nothing as the default argument.
- Useful for giving access to already-written functions from instances

```
class staticmethod( object ):
    def __init__( self, function ):
        self.function = function
    def __get__( self, client, cls ):
        """Get classmethod for class or instance"""
        return self.function
```

Ever-so-interesting staticmethod demo code...

```
class r( object ):
    def x( ):
        return 3
    x = staticmethod(x)

print r.x, r.x()
print r().x, r().x()
```

```
V:\presentations\newstyleclasses>staticmethoddescriptor.py
<function x at 0x007D2270> 3
<function x at 0x007D2270> 3
```

Python 2.4 decorator/function wrapper syntax

- Syntax supporting function wrapping
  - @callable1
@callable2
def x(*args, **named):
  - x = callable1(callable2(x))
- Soon dozens of decorator types
  - Large projects are adopting them
  - e.g. Eby is exploring multi-dispatch with metaclasses in Peak (or so rumours tell me)

Decorator/function wrapper syntax and objects...

- Keep in mind that @callable can be arbitrary callable objects, such as:
  - @withThisLock(someLock, reentrant=0)
  - @withDB(dbEnviron, newConnection=1)
  - @mustReturn((str,int,None))
  - @classmethod
  - @staticmethod
  - Any object that can accept a function (descriptor) and return a descriptor

Property descriptors

- “Managed attributes”
  - Attributes implemented with functions
  - Still looks like attribute access
  - Provide documentation mechanism
- API doesn't tell them the attribute name
  - Can't use generic implementations
  - Unless you pass in the name somehow, and then you're not using regular properties...
- Builtin “property” type

Property descriptors (cont)

- Useful when:
  - Have been using simple attributes but grow need for extra logic in one or two places and don't want to change calling pattern
  - You have readily defined functions
  - Only a few attributes need treatment
  - Working with “attributes must use method call” weanies
  - Don't feel like creating a new descriptor class

Property pseudo-code...

```
#from Raymond Hettinger's HowTo...

class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"
    def __init__(self, fget=None, fset=None, fdel=None,
doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError, "unreadable attribute"
        return self.fget(obj)
    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError, "can't set attribute"
        self.fset(obj, value)
    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError, "can't delete attribute"
        self.fdel(obj)
```

Using properties (from the documentation)

```
class C(object):
    def getx(self):
        return self.__x
    def setx(self, value):
        self.__x = value
    def delx(self):
        del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")

# writing three methods
# and a long line
# for every attribute
# gets old fast... he thinks
# not so much to himself.
```

OpenGLContext's VRML97 Fields

- Direct mapping of VRML97 fields
  - Type constrained, type-converting
  - Observable (PyDispatcher)
  - Implemented as property sub-class
    - Would do as a pure descriptor these days

```
class Coordinate( node.Node ):
    PROTO = 'Coordinate'
    point = field.newField( 'point', 'MFVec3f', 1, list)

>>> c = basenodes.Coordinate( point = [1,2,3,4,5,6] )
>>> c.point
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

Getting OpenGLContext's field properties

http://pyopengl.sourceforge.net/context/

BasicProperty descriptors

- Domain-modeling tool using Descriptors
  - Optionally-type-constrained
  - Type coercion and constraints
  - Documentable
  - Default-value/function holding
- Overridable storage
  - e.g. for PyTable RDBMS, weak-references
- Delegates work to base-type objects

Using BasicProperty descriptors...

```
from basicproperty import common, propertied, basic

class Simple( propertied.Propertied ):
    count = common.IntegerProperty(
        "count", """Count some value for us""",
        defaultValue = 0,
    )
    names = common.StringsProperty(
        "names", """Some names as a list of strings""",
    )
    mapping = common.DictionaryProperty(
        "mapping", """Mapping from name to number""",
        defaultValue = [
            ('tim',3),
            ('tom',4),
            ('bryan',5),
        ],
    )
...
```

Using BasicProperty descriptors...

```
class Simple( propertied.Propertied ):
...
    def __repr__( self ):
        className = self.__class__.__name__
        def clean( value ):
            value = value.splitlines()[0]
            if len(value) > 30:
                value = value[:27] + '...'
            return value
        props = ", ".join([
            '%s=%s'%(prop.name, repr(prop.__get__(self)))
            for prop in self.getProperties()
            if hasattr(self,prop.name)
        ])
        return '<%(className)s %(props)s>'%locals()
...
```

Using BasicProperty descriptors...

```
...
s = Simple()
s.names.append( 'this' )
s2 = s.clone( count=500 )
s.names.append( 23 )
s.names.append( u'\xf0' )
s.mapping[ 'kim' ] = 32
s.count += 1
print s
print s2
```

```
P:\properties\website\examples>simpletypes.py
<Simple count=1, mapping={'bryan': 5, 'tim': 3, 'kim': 32,
'tom': 4}, names=[u't
his', u'23', u'\xf0']>
<Simple count=500, mapping={'bryan': 5, 'tim': 3, 'kim': 32,
'tom': 4}, names=[u
'this', u'23', u'\xf0']>
```

Getting BasicProperty or more information...

http://basicproperty.sourceforge.net/

Overriding built-in types

- Many built-in functions became types
  - str, list, tuple, unicode, float, int, long
  - type, file (a.k.a. open)
- Easier customisation of functionality
  - Compared with UserDict, UserList, etc.
  - Can override normal customisation points for instances, the “slots” now map to the __*__ methods for the most part, regular methods as well, of course
- super() to access base implementation

Overriding built-in types (cont)

- Can pass sub-classes to C functions that accept only the built-in type
  - Internally the structure is the same
  - Can't generally inherit from 2 built-in types
    - Because the internal structure is different...
- For static base-types use __new__
  - __init__ comes too late
  - Signatures for e.g. list can be surprising
  - Normally must call super implementation

Subclassing list, the restricted list

You want
to create a list-like type

that calls a method
on every item to be added

to check
whether it should be allowed in the list...

Subclassing list, the restricted list

Don't argue.

Yes, you do.

Everyone wants one of these.

They're like butta, I tells ya, butta.

Subclassing list, the restricted list

- First things first, subclass as you would for any class...
- Now create an __init__ method
  - Note use of super to initialise internal struct
  - Note that we get value as a single iterable object, not as *args
- Could have used __new__, but no need

First steps to rlist.py

```
class rlist( list ):
    def __init__(self, value= None):
        """Initialize the restricted list object"""
        if value is not None:
            value = self.beforeMultipleAdd([ self.beforeAdd
(item) for item in value ])
        else:
            value = []
        super (rlist, self).__init__(value)

...

    def beforeAdd( self, value ):
        """Called before all attempts to add an item"""
        return value
    def beforeMultipleAdd( self, value ):
        """Called before attempts to add more than one item
(beforeAdd has already be called for each item)"""
        return value
```

rlist.py valiantly continues its quest for existence

- Now override all of the methods that add items (there's a lot on a list)...

```
    def __setslice__( self, start, stop, value ):
        """__setslice__ with value checking"""
        value = self.beforeMultipleAdd([ self.beforeAdd(item)
for item in value ])
        return super(rlist,self).__setslice__( start, stop,
value )
    def extend( self, value ):
        """extend with value checking"""
        value = self.beforeMultipleAdd([ self.beforeAdd(item)
for item in value ])
        return super(rlist,self).extend( value )
    __iadd__ = extend
    def append( self, value ):
        """append with value checking"""
        value = self.beforeAdd( value )
        return super(rlist,self).append( value )

# lots more here...
```

Subclassing str, or other immutable type...

You want a string that's never “”.

Just accept that you want it.

Subclassing str, the non-null string

- Only real twist is the need to use new
  - Because strings are immutable types
  - Note weird use of super!
- We can:
  - call str(self) (e.g. to get a base string implementation instead of using super)
  - pass self to C modules that require an str
- Unless we suppress it with __slots__
  - We have a dictionary

nonnullstr.py, just because we like creative names...

```
class NonNullString( str ):
    def __new__( cls, value ):
        if not str(value):
            raise ValueError( """Null string %r specified for a
"""%( str(value)) )
        else:
            return super(NonNullString,cls).__new__(cls,value)
    def __repr__( self ):
        return '%s(%r)'%(self.__class__.__name__, str(self))

if __name__ == "__main__":
    N = NonNullString( 'this' )
    print repr(N)
    try:
        NonNullString( '' )
    except ValueError:
        pass
    else:
        raise RuntimeError( """Expected ValueError""")
```

```
V:\presentations\newstyleclasses>nonnullstring.py
NonNullString('this')
```

Finally he stops blathering...

Questions?