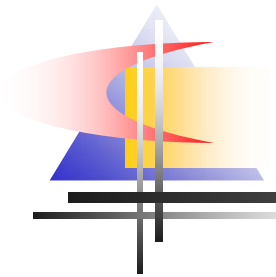
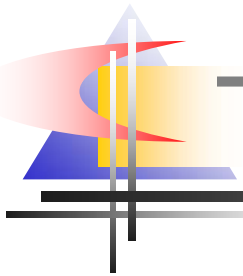


Python Metaclasses

Alex Martelli

© 2003 AB Strakt



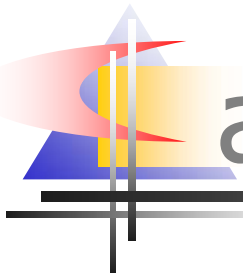
This talk's audience...:

- "fair" to "excellent" grasp of Python and OO development
- "none" to "good" grasp of Metaclasses
- wants to learn more about: Python's OO underpinnings, Metaclasses, custom Metaclasses

© 2003 AB Strakt

what's a metaclass

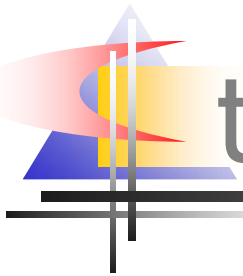
- `type(x)` is *class* (or *type*) of `x`
- `type(type(x))` is the *metaclass* (or *metatype*) of `x` (sometimes also called "the metaclass of type(x)" -- not strictly-correct usage)
- for any built-in type or new-style class `X: type(X) is type`
- for any classic-class `X: type(X) is types.ClassType`



a key book...:

- "Putting Metaclasses to Work", by Ira Forman and Scott Danforth (Addison-Wesley 1998)
- strong influence on Python 2.2 & later
- based on IBM SOMobjects & C++
- out of print, but available from used-books dealers & the like -- get it iff you're *really* keen to explore further

© 2003 AB Strakt



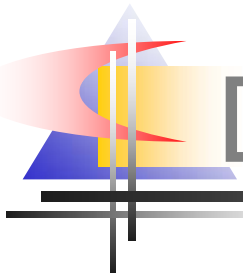
the class statement

A `class` statement is (neat, elegant syntax for) a *metaclass call* :

```
class Name[(b)]:
  # bindings into a dict d
-->
Name=metaclass('Name',b,d)
```

So -- how is *metaclass* determined...?

© 2003 AB Strakt



Determining the metaclass [0]

4-step decision procedure:

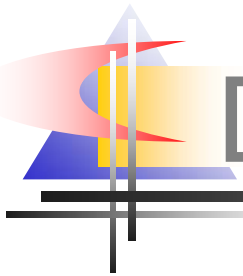
[1] explicit `__metaclass__`

[2] inherited metaclass

[3] global `__metaclass__`

[4] classic-by-default

© 2003 AB Strakt

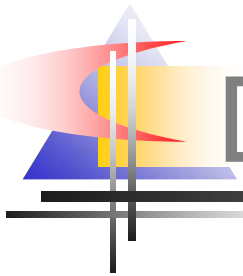


Determining the metaclass [1]

[1]: if `'__metaclass__'` is a key in d,
the corresp. value is the metaclass:

```
class Name[(b)]:
    ...
    __metaclass__ = M
-->
```

`M` is the metaclass



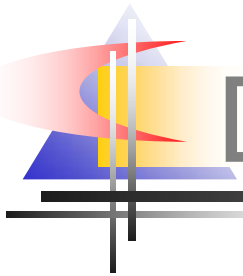
Determining the metaclass [2]

[2]: otherwise, if b is non-empty (the class has some bases), the metaclass is `type(b[0])` (type of the first base*):

```
class Name(object): pass

-->
```

the metaclass is: `type(object)` `[i.e., type]`

* there's a subtlety w/`types.ClassType`...

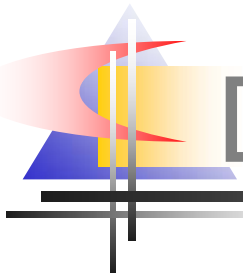


Determining the metaclass [3]

[3]: otherwise, if `__metaclass__` is a global variable, the variable's value is the metaclass:

```
__metaclass__ = M
class Name: pass
-->
```

`M` is the metaclass

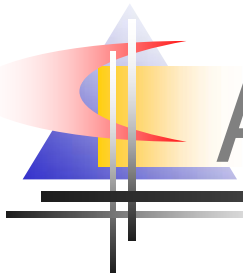


Determining the metaclass [4]

[4]: otherwise, the metaclass is `types.ClassType` ("classic-by-default")

```
class Name: pass

-->
```

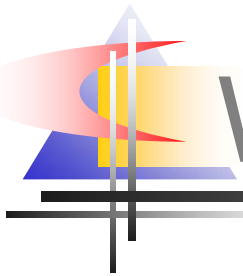
the metaclass is `types.ClassType`



A subtlety w/types.ClassType

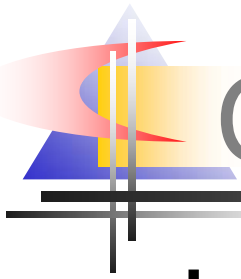
```
class X(old,new): pass
assert type(X) is type # how?!
```

- `types.ClassType`, when called, checks all bases, and delegates to the first base that's *not* a classic class
- thus: a class is classic only if *all* of its bases are classic classes



When the metaclass is called...

- it must generate an instance (normally "of itself"; normally a new one) -- just like any other call to any class or type
- calling any type proceeds through 2 steps: `__new__` then `__init__`
  - a good example of the Template Method and Factory Design Patterns in action

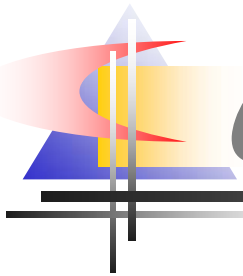


Calling a type: type.__call__

```
def __call__(cls,*a,**k):
    nu = cls.__new__(cls,*a,**k)
    if isinstance(nu, cls):
        cls.__init__(nu,*a,**k)
    return nu
```

(An example of "2-phase construction")

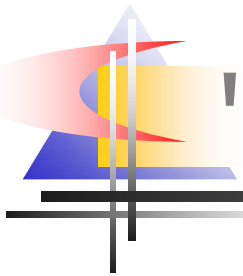
© 2003 AB Strakt



Custom metaclasses

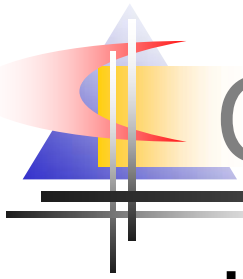
- arbitrary callables w/right signature:

```
def M(N,b,d):
   return d.get('f','%s')%N
class Ciao:
   __metaclass__ = M
   f = '*wow*%s!'
# is just like:
Ciao=M('Ciao',(),{'f':'*wow*%s!'})
# so, Ciao is a string: '*wow*Ciao!'...!
```



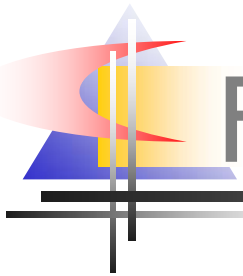
"Canonical" custom metaclasses

- normally, a metaclass `M`, when called, returns (a class that's) an instance of `M`
- just like any class: "normally, a class `C`, when called, returns an instance of `C`"
- almost invariably, canonical custom metaclasses are subclasses of **type**
- typically overriding some of `__new__`, `__init__`, `__call__`, ...



Cooperative metaclass behavior

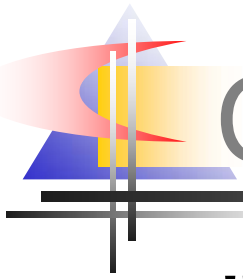
- typically overrides of `__new__`, `__init__`, `__call__`, ..., must delegate some cases to the superclass (*supermetaclass*?)
- `using type.__new__ (&c) is simplest but not cooperative`
  - I generally use it in this presentation... due to space limitations on slides!
- `consider using super instead!`



Fundamentals of Python OO

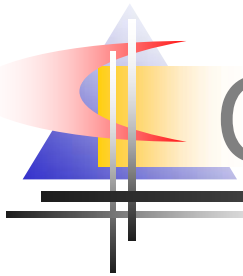
- say `type(x) is C, type(C) is M`
- `x.a` looks in `x`, else `C`, **not** up into `M`
- operations look for special methods in `C` **only** (**not** in `x`, **not** up into `M`)
- "look in `C`" always implies `C`'s own dict/descriptors then up the sequence of ancestor classes along `C`'s `MRO`

© 2003 AB Strakt



Custom metaclasses: **why**?

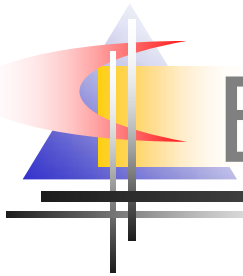
- "deeper magic than 99% of users should every worry about. If you wonder whether you need them, you don't" (Tim Peters)
- a bit pessimistic/extreme...
- worry **is** misplaced, but...
- it **is** an extra tool you **might** want...!



Custom metaclasses: **what for**?

- non-canonical uses
  - whenever you'd like `N=M('N',atuple,adict)` with `adict` handily generated with assign &c...
  - ...you **could** use `class N:` w/M as metaclass!
  - neat hack **but** -- easy to abuse -- take care!
- what can be done **only** with a CMC
  - special-methods behavior of the class itself
- what may **best** be done with a CMC
  - particularly at class-creation/calling time
  - may be simpler, faster, more general ...

© 2003 AB Strakt



Easing canonical custom-MC use

- name your metaclass 'metaSomething':

```
class metaXXX(type):
   def __new__(mcl,N,b,d):
      # e.g., alter d here
      return type.__new__(mcl,N,b,d)
```

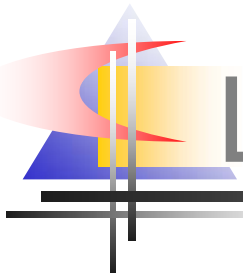
- define an auxiliary empty class:

```
class XXX: __metaclass__ = metaXXX
```

- now, classes can just inherit XXX:

```
class AnX(XXX): "...etc etc..."
```


© 2003 AB Strakt



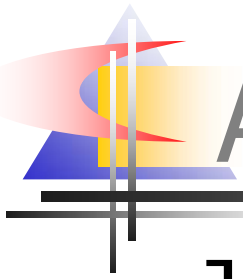
Length-handicapped vrbl nms

- a metaclass is called with 3 args:
  - *Name* of the class being instantiated
  - tuple of *Bases* for said class
  - *dictionary* with the class's attributes
- in the following, I'll write this as...:

```
class metaXXX(type):
  def __new__(mcl,N,b,d):
```

using `mcl` to stand for *metaclass* (just like `cls` for class, `self` for ordinary object) -- not widely used yet, but GvR likes it!-)

© 2003 AB Strakt

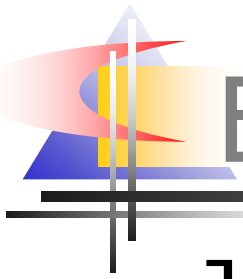


A non-canonical custom MC

```
class metaProperty(object):
  def __new__(mcl,N,b,d):
    return property(
      d.get('get'), d.get('set'),
      d.get('del'),
      d.get('__doc__') or
      'Property %s' % N)
```

or, rather equivalently:

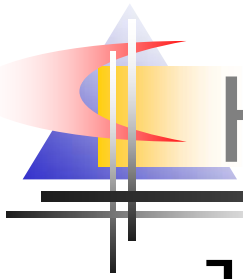
```
def metaProperty(N, b, D): ...
```



Example use of metaProperty

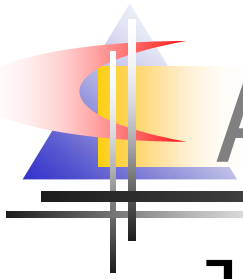
```
class SomeClass(object):
  class prop:
    __metaclass__ = metaProperty
    def get(self): return 23
  # mostly equivalent to:
  # def prop_get(self): return 23
  # prop = property(prop_get)
anobj = SomeClass()
print anobj.prop  # prints 23
```

© 2003 AB Strakt



How come this doesn't work...?

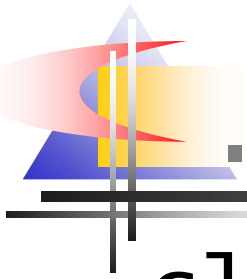
```
class Property:
  __metaclass__ = metaProperty

class SomeClass(object):
  class prop(Property):
    def get(self): return 23

# hint: what's type(Property)...?
```



A hack can make it work, e.g....:

```
class metaProperty(type):
 def __new__(mcl,N,b,d):
  if N=='Property':
    return type.__new__(mcl,N,b,d)
  else:
    return property( ...etc... )
```

Must subclass `type` and return a true instance of the metaclass for the auxiliary-class only



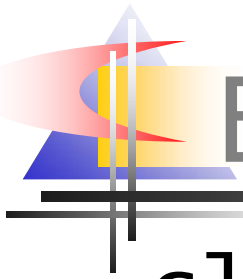
...or, an even dirtier hack...:

```
class Property: pass
Property.__class__ = metaProperty
```

Just set the `__class__` attribute of (anything...) to the metaclass... (!)

Metaclass must be any new-style class, or, for an even-dirtier sub-hack...:

```
def meta(N, b, d): ...
class Prop: pass
Prop.__class__=staticmethod(meta)
```



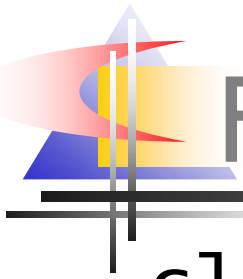
Behavior of the class object itself

```
class metaFramework(type):
  def __repr__(cls):
    return ("Framework class %r"
        % cls.__name__)
__metaclass__ = metaFramework

class XX: pass
x = XX()
print type(x)
```

© 2003 AB Strakt

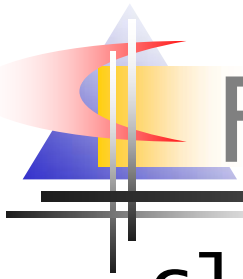
Abstract Classes

```
class metaAbst(type):
  def __call__(cls,*a,**k):
    if cls._abs:
      raise TypeError,...
    return type.__call__(...
  def __new__(mcl,N,b,d):
    d['_abs'] = (not b or not
        isinstance(b[0],metaAbst))
    return type.__new__(...
```



Final Classes [1]

```
class metaFinal(type):
  def __new__(mcl,N,b,d):
    if (b and
        isinstance(b[0],metaFinal)):
        raise TypeError, ...
    else:
      return type.__new__(mcl,N,b,d)
# unsuitable, as coded, for the
# usual class MyCl(Final): ...
# shortcut, of course
```



Final Classes [2]

```
class metaFinal(type):
  def __new__(mcl,N,b,d):
    if (b and
        isinstance(b[0],metaFinal) and
        b[0] is not Final):
        raise TypeError, ...
    else:
      return type.__new__(mcl,N,b,d)
class Final:
  __metaclass__ = metaFinal
```

© 2003 AB Strakt

Struct-like classes w/factory func

```
def struct(name, **flds):
  class st(object): pass
  st.__dict__.update(flds)
  st.__name__ = name
  def __init__(self, **afs):
    for n, v in afs.iteritems():
      if n not in flds: raise ...
      setattr(self,n,v)
  st.__dict__['__init__']=__init__
  return st
```

Struct-like classes w/CMC [1]

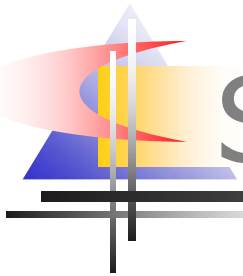
```
# reproducing the factory...:
class metaStruct(type):
  def __new__(mcl, cnm, cbs, cdf):
    def __init__(self, **afs):
      for n, v in afs.iteritems():
        if n not in cdf: raise ...
        setattr(self, n, v)
    cdf['__init__']=__init__
    return type.__new__(
        mcl, cnm, cbs, cdf)
```

© 2003 AB Strakt

Struct-like classes w/CMC [2]

```
# a metaclass allows even more...:
class metaStruct(type):
  def __new__(mcl, cnm, cbs, cdf):
    cdf['__slots__'] = cdf.keys()
    def __init__(self, **afs):
      ims=cdf.items()+afs.items()
      for n, v in ims:
        setattr(self, n, v)
    cdf['__init__'] = __init__
    ...
```

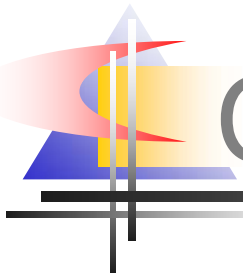
© 2003 AB Strakt



Struct-like classes w/CMC [3]

```
def __repr__(self):
  ps = []
  for n, dv in cdf.iteritems():
    v = setattr(self, n)
    if v!=dv: ps.append(repr(v))
  return '%s.%s(%s)' % (
    cdf['__module__'], cnm,
    ','.join(ps))
cdf['__repr__'] = __repr__
  ...
```

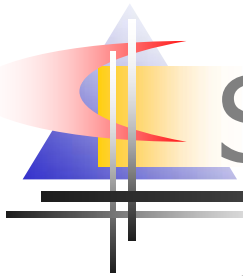
© 2003 AB Strakt



Custom metaclasses and MI

- check in `type.__new__(mc,N,b,d)`

```
class M(type): pass
class N(type): pass
class m: __metaclass__ = M
class n: __metaclass__ = N
class x(m, n): pass
TypeError: metaclass conflict;
  the metaclass of a derived
  class must be a subclass...
```

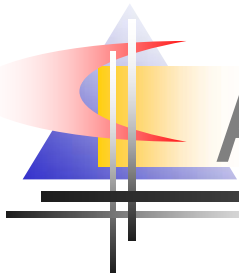


Solving metaclass conflicts

- derive custom-metaclass as needed:

```
class MN(M,N): pass
class x(m,n): __metaclass__=MN
```

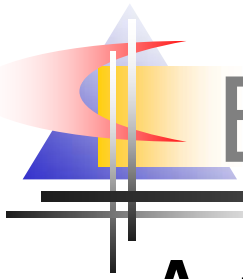
- in general, the derived metaclass must solve actual special-method conflicts
- most typical/troublesome: `__new__`
- Forman and Danforth: "inheriting metaclass constraints" (in theory, automates the derivation process)



Advanced metaclass examples

- all from Forman and Danforth's book:
  - re-dispatching
  - before/after
  - invariant-checking
  - thread-safe
  - ...
- if you're really keen on this -- get the book...!

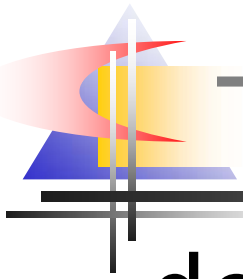
© 2003 AB Strakt



E.g.: metaTimeStamped [1]

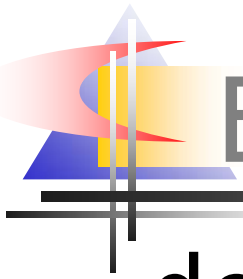
A metaclass I can "plug into" already-coded classes to make them timestamp their instances at instance-creation:

```
class mTS(type):
  def __call__(cls,*a,**k):
    x=super(mTS,cls)(*a,**k)
    x._created=time.time()
    return x
  . . .
```



TStamping w/an adjuster func

```
def addTS_to_class_init(cls):
  cinit = cls.__init__
  def init(self,*a,**k):
    cinit(self,*a,**k)
    self._created=time.time()
  cls.__init__ = cinit
# but, what if...: __slots__,
# _created conflicts, ... ?
```



E.g.: metaTimeStamped [2]

```
def __new__(mc,N,b,d):
  snm = '__slots__'
  cnm = '_created'
  sl = d.get(snm,(cnm,))
  if cnm not in sl:
    d[snm] = tuple(sl)+(cnm,)
  # ins getCreated,property,...
  return type.__new__(mc,N,b,d)
```