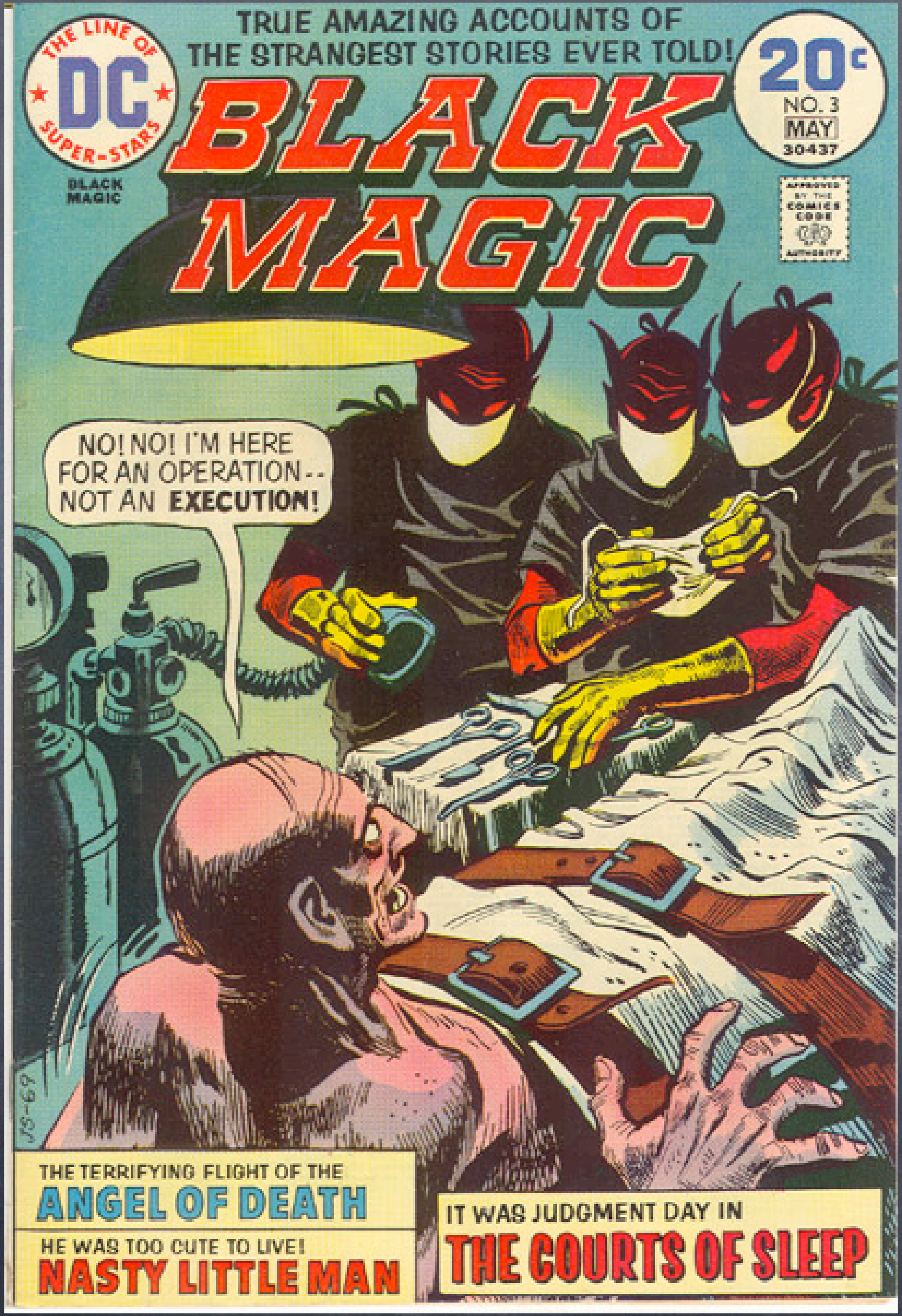
Descriptors, Decorators, Metaclasses

Python's "Black Magic"?

http://www.aleax.it/Python/osc05_bla_dp.pdf

©2005 Alex Martelli aleax@google.com


THE LINE OF DC SUPER-STARS
BLACK MAGIC
TRUE AMAZING ACCOUNTS OF THE STRANGEST STORIES EVER TOLD!
20¢
NO. 3
MAY
30437
APPROVED BY THE COMICS CODE AUTHORITY
BLACK MAGIC
NO! NO! I'M HERE FOR AN OPERATION-- NOT AN EXECUTION!
JS-69
THE TERRIFYING FLIGHT OF THE
ANGEL OF DEATH
HE WAS TOO CUTE TO LIVE!
NASTY LITTLE MAN
IT WAS JUDGMENT DAY IN
THE COURTS OF SLEEP

Descriptors

- the key infrastructure of Python's OOP
- attribute access (get, set, delete) ->
  - search class/superclasses dicts for name
  - if suitable descriptor found, delegate
- all descriptors have method __get__
- if also has __set__, data descriptor (aka override descriptor)
  - meaning: class overrides instance
- otherwise, non-data/non-override desc.

Descriptor mechanics (r)

```
x = C(); print x.foo


    ==>


if hasattr(C, 'foo'):
  d = C.foo; D = d.__class__
  if hasattr(D, '__get__') \
     and (hasattr(D, '__set__')
          or 'foo' not in x.__dict__):
    return D.__get__(d, x, C)
return x.__dict__['foo']
```

Descriptor mechanics (w)

```
x = C(); x.foo = 23


    ==>


if hasattr(C, 'foo'):
  d = C.foo; D = d.__class__
  if hasattr(D, '__set__'):
    D.__set__(d, x, 23)
    return
x.__dict__['foo'] = 23
```

Functions 'r descriptors

```
def adder(x, y): return x + y
add23 = adder.__get__(23)
add42 = adder.__get__(42)

print add23(100), add42(1000)
123 1042
```

property built-in type

property(fget=None, fset=None,
  fdel=None, doc=None)

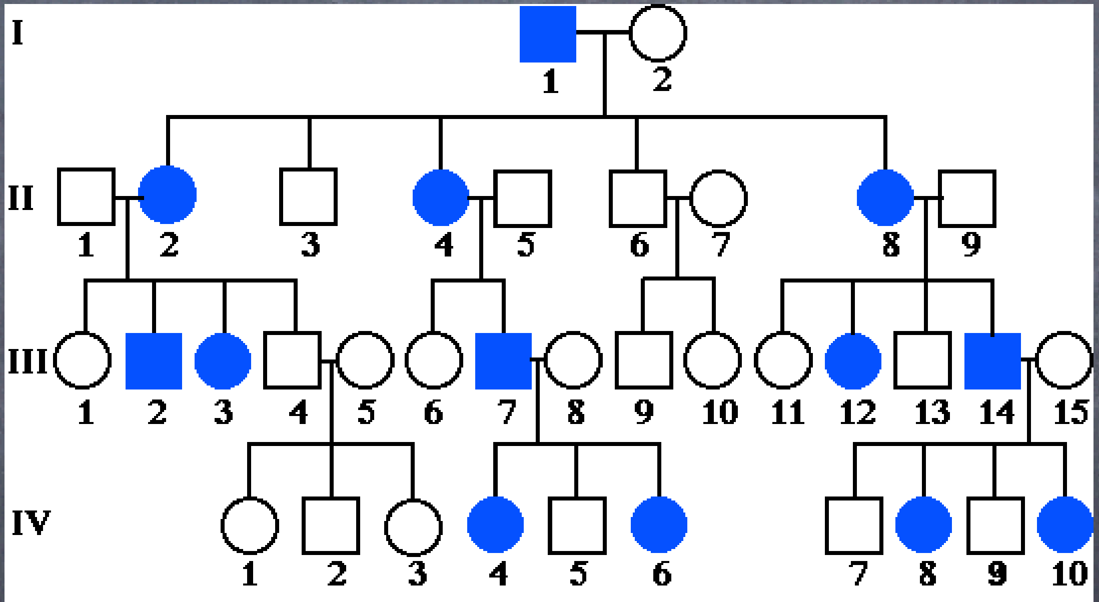
```
# fget(obj) -> value
# fset(obj, value)
# fdel(obj)
class Foo(object):
  def getBar(self): return 23
  def setBar(self, val): pass
  def delBar(self): pass
  bar = property(getBar, setBar, delBar,
    "barbarian rhubarb baritone")
```

property & inheritance

...a tricky issue w/property & inheritance:

```
class Base(object):
  def getFoo(self): return 23
  foo = property(getFoo, doc="the foo")
class Derived(Base):
  def getFoo(self): return 42

d = Derived()
print d.foo
23      # ...???
```



The extra-indirection fix

```
class Base(object):
  def getFoo(self): return 23
  def _fooget(self): return self.getFoo()
  foo = property(_fooget)
class Derived(Base):
  def getFoo(self): return 42

d = Derived()
print d.foo
```

Can be seen as a "Template Method DP"...

Custom descriptors

```
class DefaultAlias(object):
  " overridable aliased attribute "
  def __init__(self, nm): self.nm = nm
  def __get__(self, obj, cls):
    if inst is None: return self
    else: return getattr(obj, self.nm)

class Alias(DefaultAlias):
  " unconditional aliased attribute "
  def __set__(self, obj, value):
    setattr(obj, self.nm, value)
  def __delete__(self, obj):
    delattr(obj, self.nm)
```

Just-in-Time computation

```
class Jit(object):
  def __init__(self, meth, name=None):
    if name is None: name = meth.__name__
    self.meth = meth
    self.name = name
  def __get__(self, obj, cls):
    if obj is None: return self
    result = self.meth(obj)
    setattr(obj, self.name, result)
    return result
```

NB: same inheritance issues as `property`!

Decorators

- A minor syntax tweak, but...
  - syntax matters, often in unexpected ways

```
@foo
def bar(...
    ...
     ==>
def bar(...
    ...
bar = foo(bar)
```

Decorators without args

```
class X(object):
  @staticmethod
  def hello(): print "hello world"

    ==>

class X(object):
  def hello(): print "hello world"
  hello = staticmethod(hello)
```

Decorators with args

```
def withlock(L):
  def lock_around(f):
    def locked(*a, **k):
      L.acquire()
      try: return f(*a, **k)
      finally: L.release()
    locked.__name__ = f.__name__
    return locked
  return lock_around
class X(object):
  CL = threading.Lock()
  @withlock(CL)
  def amethod(self):
    ...
```

Always-fresh defaults

```
def always_fresh_defaults(f):
  from copy import deepcopy
  defaults = f.func_defaults
  def refresher(*a, **k):
    f.func_defaults = deepcopy(defaults)
    return f(*a, **k)
  return refresher
@always_fresh_defaults
def packitem(item, pack=[]):
  pack.append(item)
  return pack
```

A property-packager

```
def prop(f): return property(*f())

class Rectangle(object):
  def __init__(self, x, y):
    self.x, self.y = x, y
  @prop
  def area():
    def get(self): return self.x*self.y
    def set(self, v):
      ratio = math.sqrt(v/self.area)
      self.x *= ratio
      self.y *= ratio
    return get, set
```

A generic decorator

```
def processedby(hof):
  def processedfunc(f):
    def wrappedfunc(*a, **k):
      return hof(f, *a, **k)
    wrappedfunc.__name__ = f.__name__
    return wrappedfunc
  return processedfunc

# e.g, hof might be s/thing like...:
def tracer(f, *a, **k):
  print '%s(%s,%s)' % (f, a, k),
  r = f(*a, **k)
  print '->', repr(r)
  return r
```

Metaclasses

- Every class statement uses a metaclass
  - mostly `type` (or `types.ClassType`)
  - no hassle, no problem, no issue
- but, you can make a custom metaclass

Semantics of class

```
class X(abase, another):
  ...classbody...
  ==>
1. execute ...classbody... building a dict D
2. identify metaclass M
  2.1 __metaclass__ in D
  2.2 leafmost metaclass among bases
    (metatype conflict may be diagnosed)
  2.3 if no bases, __metaclass__ global
  2.4 last ditch, types.ClassType (old!)
3. X = M('X', (abase, another), D)
  in whatever scope for class statement
```

Make a class on the fly

```
# just call the metaclass...:
bunch = type('bunch', (),
          dict(foo=23, bar=42, baz=97))

# is like...:

class bunch(object):
  foo = 23
  bar = 42
  baz = 97
# but may be handier (runtime-given name,
# bases, and/or dictionary...).
```

A tiny custom metaclass

```
class MetaRepr(type):
  def __repr__(cls):
    C = cls.__class__    # leaf metaclass
    N = cls.__name__
    B = cls.__bases__
    D = cls.__dict__
    fmt = '%s(%r, %r, %r)'
    return fmt % (C, N, B, D)
```

Make accessor methods

```
class M(type):
  def __new__(cls, N, B, D):
    for attr in D.get('__slots__', ()):
      if attr.startswith('_'):
        def get(self, attr=attr):
          return getattr(self, attr)
        get.__name__ = 'get' + attr[1:]
        D['get' + attr[1:]] = get
    return super(M, cls).__new__(N, B, D)
class Point:
  __metaclass__ = M
  __slots__ = '_x', '_y'
```

Tracking instances

```
from weakref import WeakValueDictionary
class mIT(type):
  def __init__(cls, N, B, D):
    super(mIT, cls).__init__(N, B, D)
    cls._inst = WeakValueDictionary()
    cls._numcreated = 0
  def __call__(cls, *a, **k):
    inst = super(mIT, cls)(*a, **k)
    cls._numcreated += 1
    cls._inst[cls._numcreated] = inst
    return inst
  def __instances__(cls):
    return cls._inst.values()
```

Metatype conflicts

```
class meta_A(type): pass
class meta_B(type): pass
class A: __metaclass__ = meta_A
class B: __metaclass__ = meta_B

class C(A, B): pass

# no leafmost metaclass -> conflict

# manual metatype conflict resolution:
class meta_C(meta_A, meta_B): pass
class C(A, B): __metaclass__ = meta_C
```

MC Conflict Resolution

We can automate it...

Removing redundancy

```
def uniques(sequence, skipset):
  for item in sequence:
    if item not in skipset:
      yield item
      skipset.add(item)

def no_redundant(classes):
  reds = set([types.ClassType])
  for c in classes:
    reds.update(inspect.getmro(c)[1:])
  return tuple(uniques(classes, reds))
```

Build "no conflict" metaclass

```
def _noconf(bases, L, R):
  metas = L + tuple(map(type,bases)) + R
  metas = no_redundant(metas)
  if not metas: return type
  elif len(metas)==1: return metas[0]
  for m in metas:
    if not issubclass(m, type):
      raise TypeError, 'NTR %r' % m
  n = '_'.join(m.__name__ for m in metas)
  return classmaker()(n, metas, {})
```

The class-maker closure

```
def classmaker(L=(), R=()):
  def maker(n, bss, d):
    return _noconf(bss, L, R)(n, bss, d)
  return maker
```

The mutual recursion between _noconf and classmaker ensures against possible conflicts at any meta-level (metacl of metacl of ...)

Using noconf.py

```
class meta_A(type): pass
class meta_B(type): pass
class A: __metaclass__ = meta_A
class B: __metaclass__ = meta_B

import noconf
class C(A, B):
  __metaclass__ = noconf.classmaker()
```