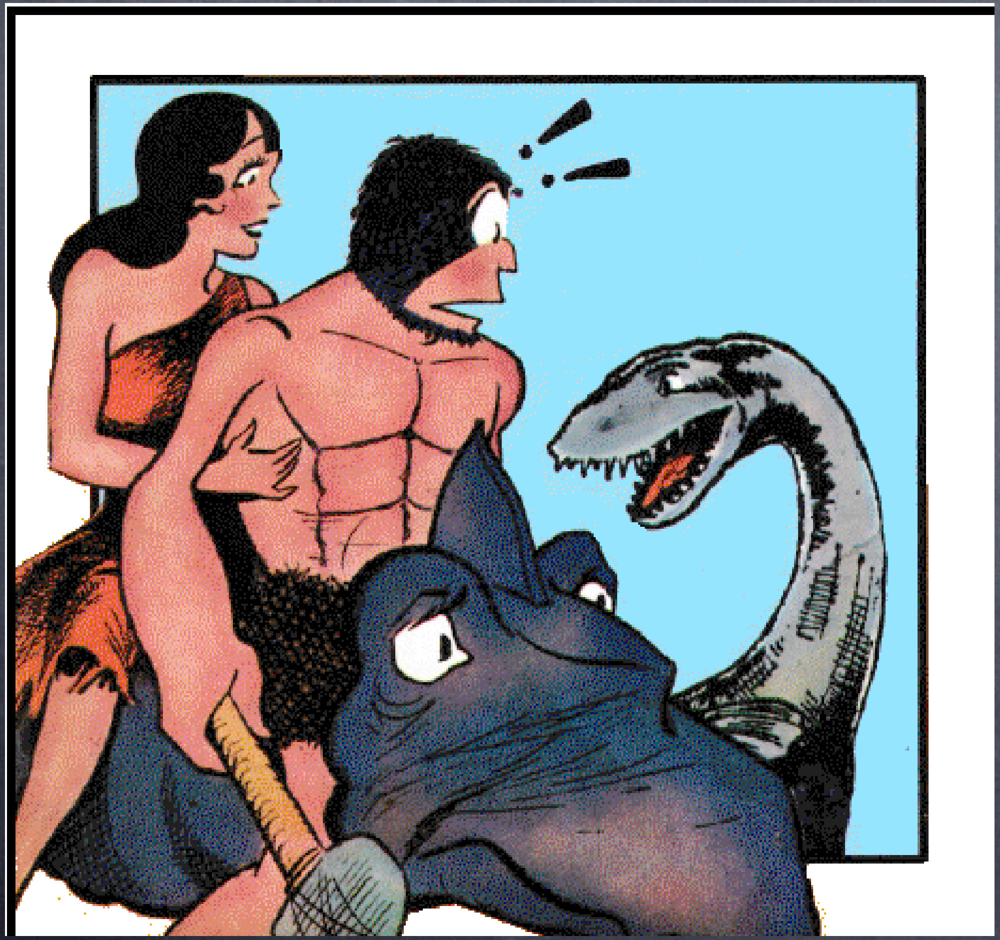
Python's Object Model

Objects by Design

http://www.aleax.it/Python/nylug05_om.pdf

What's OOP?

I dunno -- what's OOP with you?



Alley Oop...?

Three faces of OOP

- OOP: package state and behavior into suitable "chunks", in order to achieve...:
- Delegation
  - let something else do (most of) the work
- Polymorphism
  - act "as if" you were something else
- Instantiation
  - one "blueprint", many instances

OOP for delegation

- intrinsic/implicit (via attribute lookup):
  - instance -> class
  - class -> descriptors
  - class -> base classes
- overt/explicit:
  - containment and delegation (hold/wrap)
  - delegation to self
- inheritance: more rigid; IS-A...
- hold/wrap: more flexible; USES-A...

Attribute lookup

- x.y [and identically x.y()!] means:
  - check out descriptor stuff
  - or else try x.__dict__['y']
  - or else try type(x).__dict__['y']
  - or else try:
    for base in type(x).__mro__: ...
- x.y = z means:
  - check out descriptor stuff
  - or else x.__dict__['y'] = z

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(); return 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'] # or from C, &c
```

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 are 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)

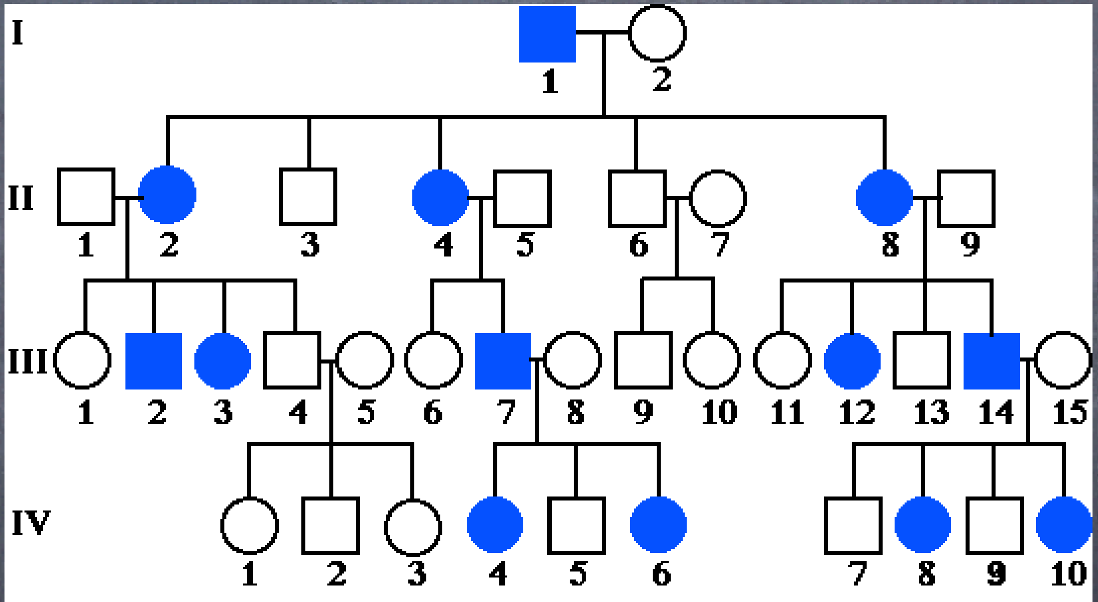
```
# fget(obj) -> value
# fset(obj, value)
# fdel(obj)
class Foo(object):
  def getBar(self): return 23
  def setBar(self, val): pass
  def delBar(self): pass
  bar = property(getBar, setBar, delBar,
    "barbarian rhubarb baritone")
```

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 obj 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`!

Pydioms: hold vs wrap

- "Hold": object O has subobject S as an attribute (maybe property) -- that's all
  - use self.S.method or O.S.method
  - simple, direct, immediate, but coupling on the wrong axis
- "Wrap": hold (often via private name) plus delegation (so you use O.method)
  - explicit (def method(self...)...self.S.method)
  - automatic (delegation in __getattr__)
  - gets coupling right (Law of Demeter)

Wrapping to restrict

```
class RestrictingWrapper(object):
  def __init__(self, w, block):
    self._w = w
    self._block = block
  def __getattr__(self, n):
    if n in self._block:
      raise AttributeError, n
    return getattr(self._w, n)
  ...
```

Inheritance cannot restrict!

But...: special methods require special care

Self-delegation == TMDP

- Template Method design pattern
- great pattern, lousy name
  - the word "template" is way overloaded...!
- classic version:
  - abstract base's organizing method...
  - ...calls hook methods of subclasses
  - client code calls OM on instances
- mixin version:
  - mixin base's OM, concrete classes' hooks

TMDP in Queue.Queue

```
class Queue:
  ...
  def put(self, item):
    self.not_full.acquire()
    try:
      while self._full():
        self.not_full.wait()
      self._put(item)
      self.not_empty.notify()
    finally:
      self.not_full.release()
  def _put(self, item):
    self.queue.append(item)
```

Queue's TMDP

- Not abstract, most often used as-is
  - so, must implement all hook-methods
- subclass can customize queueing discipline
  - with no worry about locking, timing, ...
  - default discipline is simple, useful FIFO
  - can override hook methods (_init, _qsize, _empty, _full, _put, _get) AND also...
  - ...data (maxsize, queue), a Python special

Customizing Queue

```
class LifoQueue_with_deque(Queue):
  def _put(self, item):
    self.queue.appendleft(item)

class LifoQueue_with_list(Queue):
  def _init(self, maxsize):
    self.maxsize = maxsize
    self.queue = list()
  def _get(self):
    return self.queue.pop()
```

DictMixin's TMDP

- Abstract, meant to multiply-inherit from
  - does not implement hook-methods
- subclass must supply needed hook-methods
  - at least __getitem__, keys
  - if R/W, also __setitem__, __delitem__
  - normally __init__, copy
  - may override more (for performance)

TMDP in DictMixin

```
class DictMixin:
  ...
  def has_key(self, key):
    try:
      # implies hook-call (__getitem__)
      value = self[key]
    except KeyError:
      return False
    return True
  def __contains__(self, key):
    return self.has_key(key)
  # NOT just: __contains__ = has_key
```

Chaining Mappings

- given multiple mappings (e.g. dictionaries) in a given order,
- we want to build a “virtual” read-only mapping by chaining the given dicts
- i.e., try each lookup on each given dict, in order, until one succeeds or all fail

Exploiting DictMixin

```
class Chainmap(UserDict.DictMixin):
  def __init__(self, mappings):
    self._maps = mappings
  def __getitem__(self, key):
    for m in self._maps:
      try: return m[key]
      except KeyError: pass
    raise KeyError, key
  def keys(self):
    keys = set()
    for m in self._maps: keys.update(m)
    return list(keys)
```

State and Strategy DPs

- Somewhat like a "Factored-out" TMDP
  - OM in one class, hooks in others
  - OM calls self.somedelegate.dosomehook()
- classic vision:
  - Strategy: 1 abstract class per decision, factors out object behavior
  - State: fully encapsulated, strongly coupled to Context, self-modifying
- Python: can also switch __class__, methods

Strategy DP

```
class Calculator(object):

  def __init__(self):
    self.setVerbosity()

  def setVerbosity(self, quiet=False):
    if quiet: self.strat = Quiet()
    else: self.strat = Show()

  def compute(self, expr):
    res = eval(expr)
    self.strat.show('%r=%r'% (expr, res))
```

Strategy classes

```
class Show(object):
  def show(self, s):
    print s


class Quiet(Show):
  def show(self, s):
    pass
```

State DP: base class

```
class Calculator(object):

  def __init__(self):
    self.state = Show()

  def compute(self, expr):
    res = eval(expr)
    self.state.show('%r=%r'% (expr, res))

  def setVerbosity(self, quiet=False):
    self.state.setVerbosity(self, quiet)
```

State classes

```
class Show(object):

  def show(self, s):
    print s

  def setVerbosity(self, obj, quiet):
    if quiet: obj.state = Quiet()
    else: obj.state = Show()

class Quiet(Show):

  def show(self, s):
    pass
```

Ring Buffer

- FIFO queue with finite memory: stores the last MAX (or fewer) items entered
  - good, e.g., for logging purposes
- intrinsically has two macro-states:
  - early (<=MAX items entered yet), just append new ones
  - later (>MAX items), each new item added must overwrite the oldest one remaining (to keep latest MAX items)
- switch from former macro-state (behavior) to latter is massive, irreversible

Switching __class__ (1)

```
class RingBuffer(object):
  def __init__(self, MAX=256):
    self.d = list()
    self.MAX = MAX
  def tolist(self):
    return list(self.d)
  def append(self, item):
    self.d.append(item)
    if len(self.d) == self.MAX:
      self.c = 0
      self.__class__ = _FullBuffer
```

Switching __class__ (2)

```
class _FullBuffer(object):
  def append(self, item):
    self.d[self.c] = item
    self.c = (1+self.c) % self.MAX
  def tolist(self):
    return ( self.d[self.c:] +
             self.d[:self.c] )
```

Switching a method

```
class RingBuffer(object):
  def __init__(self, MAX=256):
    self.d = list()
    self.MAX = MAX
  def append(self, item):
    self.d.append(item)
    if len(self.d) == self.MAX:
      self.c = 0
      self.append = self._append_full
  def _append_full(self, item):
    self.d.append(item)
    self.d.pop(0)
  def tolist(self): return list(self.d)
```

OOP for polymorphism

- intrinsic/implicit/classic:
  - inheritance (single/multiple)
- overt/explicit/pythonic:
  - adaptation and masquerading DPs
  - special-method overloading
  - advanced control of attribute access
  - custom descriptors (and metaclasses)

Python's polymorphism

- ...is notoriously based on duck typing...:

(why a duck?)

Restricting attributes

```
class Rats(object):
  def __setattr__(self, n, v):
    if not hasattr(self, n):
      raise AttributeError, n
    super(Rats, self).__setattr__(n, v)
```

affords uses such as:

```
class Foo(Rats):
  bar, baz = 1, 2
```

so no new attributes can later be added.
None of __slots__'s issues (inheritance &c)!

So, ___slots___ or Rats?

(IF needed at all ...):

`__slots__` strictly, only to save memory

(classes with LOTS of tiny instances)

Rats (& the like) for everything else

Remember *AGNI*

Ain't Gonna Need It!

class instance as module

```
class _const(object):
  class ConstError(TypeError): pass
  def __setattr__(self, n, v):
    if n in self.__dict__:
      raise self.ConstError, n
    super(_const, self).__setattr__(n, v)
import sys
sys.modules[__name__] = _const()
```

here, no **existing** attributes can be **changed**

Functor or closure?

```
class Functor(object):
  def __init__(self, init args):
    ...set instance data from init args...
  def __call__(self, more args):
    ...use instance data and more args...

def outer(init args):
  ...set local vars (if needed) from init args...
  def inner(more args):
    ...use outer vars and more args...
  return inner
```

"closure factory" is simpler!

Closure or functor?

```
class SubclassableFunctor(object):
  def __init__(self, init args):
    ...set instance data from init args...
  def do_hook1(self, ...): ...
  def do_hook2(self, ...): ...
  def __call__(self, more args):
    ...use instance data and more args
      and call hook methods as needed...
```

class is more powerful and flexible, since subclasses may easily customize it

use only as much power as you need!

specials come from class

```
def restrictingWrapper(w, block):
  class c(RestrictingWrapper): pass
  for n, v in get_ok_specials(w, block):
    def mm(n, v):
      def m(self, *a, **k):
        return v(self._w, *a, **k)
      return m
    setattr(c, n, mm(n, v))
  return c(w, block)
```

get_ok_specials

```
import inspect as i
def get_ok_specials(w, block):
  """ skip nonspecial names, ones in
      `block` or in RestrictingWrapper,
      and '__getattribute__' """
  for n, v in i.getmembers(
      w.__class__, i.ismethoddescriptor):
    if (n[:2] != '__' or n[-2:] != '__'
       or n in block or
       n == '__getattribute__' or
       n in RestrictingWrapper.__dict__):
      continue
    yield n, v
```

OOP for instantiation

- one class -> many instances
  - same behavior, but distinct state
  - per-class behavior, per-instance state
- ...but at (rare) times we don't want that...
  - while still requiring other OOP feechurz
  - thus: Singleton (forbid "many instances")
  - or: Monostate (remove "distinct state")

Singleton ("Highlander")

```
class Singleton(object):
  def __new__(cls, *a, **k):
    if not hasattr(cls, '_inst'):
      cls._inst = super(Singleton, cls
               ).__new__(cls, *a, **k)
    return cls._inst
```

subclassing is a nasty problem, though:

```
class Foo(Singleton): pass
class Bar(Foo): pass
f = Foo(); b = Bar(); # ...???...
```

this problem is intrinsic to any Singleton!

Monostate ("Borg")

```
class Borg(object):
  _shared_state = {}
  def __new__(cls, *a, **k):
    obj = super(Borg, cls
             ).__new__(cls, *a, **k)
    obj.__dict__ = cls._shared_state
    return obj
```

subclassing no problem, data override helps:

```
class Foo(Borg): pass
class Bar(Foo): pass
class Baz(Foo): _shared_state = {}
```