DP and OOP in Python

Objects by Design

©2005 Alex Martelli aleaxit@gmail.com
What’s OOP?

I dunno -- what’s OOP with you?

Alley Oop

Alley Oop...?
OOP as delegation

intrinsic/implicit:
- 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...
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!
However...: what about special methods?
Self-delegation == TMDP

Template Method design pattern
great pattern, lousy name
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
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)
    ...

TMDP in Queue.Queue
Queue’s TMDP

- Not abstract, 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
- could override hook methods (_init, _qsize, _empty, _full, _put, _get) AND...
- ...data (maxsize, queue), a Python special
Customizing Queue

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

class LifoQueueB(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)
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

- Not unlike 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 switch __class__, methods
class Calculator(object):
    def __init__(self):
        self.strat = Show()
    def compute(self, expr):
        res = eval(expr)
        self.strat.show('%r=%r' % (expr, res))
    def setVerb(self, quiet=False):
        if quiet: self.strat = Quiet()
        else: self.strat = Show()

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

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

class Calculator(object):
    def __init__(self):
        self.state = Show()
    def compute(self, expr):
        res = eval(expr)
        self.state.show('%r=%r' % (expr, res))
    def setVerb(self, quiet=False):
        self.state.setVerb(self, quiet)

class Show(object):
    def show(self, s):
        print s
    def setVerb(self, obj, quiet):
        if quiet:
            obj.state = Quiet()
        else:
            obj.state = Show()

class Quiet(Show):
    def show(self, s):
        pass
class RingBuffer(object):
    class _Full(object):
        def append(self, item):
            self.d[self.c] = item
            self.c = (1+self.c) % MAX
        def tolist(self):
            return self.d[self.c:]+self.d[:self.c]
    def __init__(self):
        self.d = []
    def append(self, item):
        self.d.append(item)
        if len(self.d) == MAX:
            self.c = 0
            self.__class__ = self._Full
    def tolist(self):
        return list(self.d)
class RingBuffer(object):
    def __init__(self): self.d = []
    def append_full(self, self, item):
        self.d.append(item)
        self.d.pop()
    def append(self, self, item):
        self.d.append(item)
        if len(self.d) == MAX:
            self.c = 0
            self.append = self.append_full
    def tolist(self, 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 bound. None of __slots__'s issues (inheritance &c)!
So, `__slots__` or Rats?

`__slots__` strictly, only to save memory classes with LOTS of tiny instances.

Rats (& the like) for everything else.

(if needed at all... remember *AGNI*!)

20
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.module[__name__] = _const()
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)

def get_ok_specials(w, block):
    'use inspect\'s getmembers and ismethoddescriptor, skip nonspecial names, ones in block, ones already in RestrictingWrapper, __getattribute__'
import inspect as i

def get_ok_specials(w, block):
    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
Null Object DP

- instead of None, an object "innocuously polymorphic" with any expected objects
- "implement every method" to accept arbitrary arguments and return self
- special methods need special care
- advantage: avoid many "if x is None:" tests
- or other similar guards
A general Null class

class Null(object):
    def __init__(self, *a, **k): pass
    def __call__(self, *a, **k):
        return self
    def __repr__(self): return 'Null()'
def __len__(self): return 0
def __iter__(self): return iter(())
__getattr__ = __call__
__setattr__ = __call__
__delattr__ = __call__
__getitem__ = __call__
__setitem__ = __call__
__delitem__ = __call__
A specialized Null class

class NoLog(object):
    def write(self, data): pass
    def writelines(self, self, data): pass
    def flush(self): pass
    def close(self): pass

either class allows:
if mustlog: logfile = file(...) else: logfile = Null() # or NoLog()
then throughout the code, just
logfile.write(xx) # no guard 'if logfile'
specialized version may detect more errors
OOP for instantiation

- one class -> many instances
- same behavior, but distinct state
- per-class behavior, per-instance state

...but sometimes we don't want that...

while still requiring other OOP thingies

thus: Singleton (forbid "many instances")

or: Monostate (remove "distinct state")
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 problem, though:
class Foo(Singleton): pass
class Bar(Foo): pass
f = Foo(); b = Bar(); # ...???
problem is intrinsic to Singleton
class Callable(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 from init args
    def inner(more args):
        use outer vars and more args
    return inner

"closure factory" is simpler!
Closure or class?

class CallableSubclassable(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, as subclasses may easily customize

use only the power you need!
Monostate ("Borg")

```python
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 is no problem, just:
```python
class Foo(Borg): pass
class Bar(Foo): pass
class Baz(Foo): _shared_state = {}  
data overriding to the rescue!
```