Python's Meta-Object Protocol

Guido van Rossum
guido@python.org

9th LASER summer school, Sept. 2012
What's a Meta-Object Protocol?

- Some fancy word I picked up from a book
    - by Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow
  - *Putting Metaclasses to Work* (1998)
    - by Ira R. Forman and Scott H. Danforth
    - (this is the book I actually read :-)

- Runtime manipulation of types/classes
- More powerful than mere introspection
- Idea apparently originated in Common Lisp
- Introduced in Python in several stages
Early Python versions (pre–2001) had no MOP
There was plenty of introspection though
  ◦ E.g. obj.__class__, obj.__dict__, cls.__dict__
C code could create new types
  ◦ these were not the same as user–defined classes
Enabling feature: no constructor syntax
  ◦ instance construction is just class invocation
    • C(x, y) creates a C instance c and calls c.__init__(x, y)
Seed idea: "Don Beaudry Hook"
The "Don Beaudry Hook" (1995)

- Starting point: class declaration machinery
  - class Foo(BaseClass):
    - def bar(self, arg): ... # method definitions
  - Invokes an internal API to construct the class Foo
  - Foo = <API>('Foo', (BaseClass,), {'bar': ...})
- Don's proposed tweak:
  - If BaseClass's type is callable, call it instead
- Don successfully lobbied for this feature
- Zope's ExtensionClass made it popular
What is a Python Class Really?

- Class statement is mostly a runtime thing
- Syntax: "class" <name> <bases> ":" <suite>
- Runtime:
  - evaluate <bases> into a tuple
  - evaluate <suite> into a dict (capture locals)
  - <name> = SomeAPI(name, bases, suite_dict)
- Don's hook determines SomeAPI from bases
- SomeAPI() can return whatever it wants
- This gives the base class total control!
This Was Just the Beginning

- Don's original hook required writing C code
- In 1999, I added pure Python support
  - SomeAPI == bases[0].__class__, if it exists
  - This was so head-exploding at the time, the essay was nicknamed "The Killer Joke"

- First time the term metaclass was used:
  - bases[0].__class__ == Foo.__class__ == metaclass
  - "the class of the class"

- But there wasn't much metaclass protocol
Proper Metaclasses (2001)

- typeobject.c grew from 50 to 5000 lines!
- Inspired by reading Kiczales c.s.
- Generalization: D.B. Hook is always used
- Every class has a __class__ attribute
- Simpler spelling: __metaclass__ = ...
- Default: __class__ of first base class whose __class__ isn't the default (built-in) metaclass
- Default default: create a "classic" class
- Use "class Foo(object): ..." for "new-style"
Much More Changed...

- Builtins like `int()`, `str()` became types/classes
- `bool` (a bit of an embarrassment, actually)
- Unified built-in types, user-defined classes
  - (well, mostly; some restrictions still apply)
- Improved multiple inheritance, `super()`
  - MRO changed from depth-first to sensible
  - (improved again in 2003, adopting C3 algorithm)
- Construction of immutable objects: `__new__()`
- Descriptors
- Slots
A great many conversion functions existed
  ◦ e.g. int(), float(), str(), list(), tuple()

We changed all these to become classes
  ◦ Also dict, set, bool
  ◦ And in Python 3: range, bytes, bytearray
  ◦ (In Python 2: long, unicode; gone in Python 3)

Special case: type() is overloaded on #args:
  ◦ type(classname, bases, localsdict) creates a new class from its arguments
  ◦ type(x) returns the type of x (usually x.__class__)
The Boolean Type

- The type bool was added to Python 2.3
- However the constants False and True had been added to 2.2.1 (with values 0 and 1)
  - IOW Python 2.2[.0] did not have False/True
  - This violated our own compatibility rules!
  - Mea Culpa — won't happen again!
- Other bool peculiarities:
  - bool subclasses int; you can't subclass bool
  - False == 0; True == 1; True + True == 2
  - False and True are the only two instances
Goal: subclass built-in types; e.g.:
- class casedict(dict):
  - def __setitem__(self, key, val):
    - dict.__setitem__(self, key.lower(), val)

In practice need to override many methods

Still, useful to add new methods; e.g.:
- class url(str):
  - def parse(self): return urllib.parse.urlparse(self)
  - def __new__(cls, arg=''):  
    - if isinstance(arg, tuple):
      - arg = urllib.parse.urlunparse(arg)
    - return super().__new__(cls, arg)
Method Resolution Order

- Order in which base class dicts are searched
  - This matters for multiple inheritance
  - MRO is a misnomer; it's used for all attributes
- Example ("diamond" order):
  - class A; class B(A); class C(A); class D(B, C)
- Old MRO: depth first: D, B, A, C, A
- Python 2.2: ditto with twist: D, B, C, A
- Python 2.3 and later: C3: D, B, C, A
  - Comes from Dylan (a Lisp spin-off)
  - Better in some more complicated cases
Why Does C3 Matter?


- **Local precedence order**
  - "Order of direct subclasses should be preserved"
  - Ergo, if C(X, Y), then X before Y in C.mro()

- **Monotonicity**
  - "A subclass should not reorder MRO of bases"
  - IOW if X before Y in C.mro(), and D(C), then X before Y in D.mro()

- **For examples, read the 2.3 paper**
  - Also, if C(X, Y) and D(Y, X), then E(C, D) is an error
Super Calls

- Python 1 through 2.1 syntax:
  - class MyClass(Base):
    - def mycall(self, arg):
      - Base.mycall(self, arg)

- Python 2.2 syntax:
  - super(MyClass, self).mycall(arg)

- Python 3 syntax:
  - super().mycall(arg)

- Python 4 syntax:
  - ???
Cooperative Super Calls

- Why introduce super()?
- Diamond diagram again:
  - class A: def dump(self): print(...)
  - class B(A): def dump(self): print(...); A.dump(self)
  - class C(A): def dump(self): print(...); A.dump(self)
  - class D(B, C): def dump(self): print(...); ???
    - Wants to call B.dump(), C.dump(), A.dump()
    - EACH EXACTLY ONCE, IN THAT ORDER
    - Prefer not to have to modify B or C
    - D shouldn't have to know about A at all

- This is why you need "super" built in
Making Immutable Objects

- `__init__()` is called after object is constructed
  - Ergo it can only mutate an existing object
  - How to subclass e.g. int, str or tuple?

- Hack:
  - mark object as immutable afterwards

- Better:
  - `__new__()` constructor returns a new object
  - `__new__()` is a class method
  - `__new__(cls)` must call `super().__new__(cls)`

- System calls `__new__()` followed by `__init__()`
Descriptors

- Generalization of method binding. Example:
  - class Foo:
    - def bar(self, label): print(label, id(self))
  - "bar" is a plain function with two arguments
  - Yet after x = Foo(), we can call x.bar('here')
  - The magic is all in "x.bar"
  - It returns a "bound method":
    - Short-lived (usually) helper object
    - Points to x and to bar (the plain function)
Generalizing Bound Methods

Where do bound method objects come from?

Special case instance attribute lookup:
1. look in instance dict
2. look in class dict
3. look in base class dicts (in "MRO" order)

In steps 2–3, if the value is a function, construct a bound method

Generalization: ask the object if it can construct a bound method: obj.__get__(...)

Here __get__ is part of descriptor protocol
Use Cases For Descriptors

- Improved way to define computed attributes:
  - class C:
    ```python
    ...
    @property
    def foo(self): return <whatever>
    ```

- Static and class methods:
  - class C:
    ```python
    @staticmethod
    def foo(): return <anything>
    @classmethod
    def foo(cls): return <something>
    ```
Data Descriptors

- On attribute assignment:
  1. look in class dict
  2. look in base class dicts (in MRO order)
  3. store in instance dict
- In steps 1–2, if the object found has a `__set__` method, call it (and stop)
- Note that step 3 is last!
  - Otherwise the other steps would never be used
Slots

- Special use of data descriptors; syntax:
  - class C:
    __slots__ = ['foo', 'bar']
- This auto-generates data descriptors
- *And* allocates space in the object
- *And* skips adding a __dict__ to the object
  - (unless a base class already defines __dict__) 
- Use cases:
  - Reduce memory footprint of instances
  - Disallow accidental assignment to other attributes
Small Tweaks in Python 3

- Reference: PEP 3115
- Improved syntax to set the metaclass:
  - class Foo(base1, ..., metaclass=FooMeta): ...
  - (this is needed to enable the next feature)
- metaclass can override dict type for <suite>:
  - use case: record declaration order in OrderedDict
  - @classmethod
def __prepare__(cls, name, bases, **kwds):
    return dict() # Or some subclass of dict
  - <suite> is executed in this dict (subclass)
Object:
- reference count
- type pointer
- slots
- one of the slots may be a dict

Type (derives from object):
- specific slots:
  - list of methods
  - list of slot descriptors
- type of type → itself