Advanced Python Objects

Reuven M. Lerner, PhD
reuven@lerner.co.il

__add__

- Add two elements together
- __add__ gets two parameters
  - self
  - Another object
- How do you add them? That's up to you!
- Typically, you'll return a new instance of the same class

Example

```
class Foo(object):
    def __init__(self, x):
       self.x = x
    def __add__(self, other):
       return Foo(self.x + other.x)
    def __repr__(self):
      return "Instance of f, x = {}".format(self.x)


f1 = Foo(10)
f2 = Foo(20)
print(f1 + f2)
```

__iadd__

- We can also address what happens when += is invoked
- This is not the same as __add__!
- The expectation is that you'll change the state of the current object, and then return self.

Example

```
class Foo(object):
    def __init__(self, x):
       self.x = x
    def __iadd__(self, other):
      self.x += other.x
      return self
    def __repr__(self):
       return "Instance of f, x = {}".format(self.x)


f1 = Foo(10)
f2 = Foo(20)
f1 += f2
print("Now f1 = {}".format(f1))
```

Making __add__ flexible

- Remember, the second argument can be anything
- If we want, we can play games with that — checking for attributes

Example

```
class Foo(object):
    def __init__(self, x):
        self.x = x
    def __add__(self, other):
        if hasattr(other, 'x'):
            return Foo(self.x + other.x)
        else:
            return Foo(self.x + other)
   def __repr__(self):
        return "Instance of f, x = {}".format(self.x)


f1 = Foo(10)
f2 = Foo(20)
print("f1 + f2 = {}".format(f1 + f2))
print("f1 + 10 = {}".format(f1 + 50))
```

Reversible?

- What if we now say

```
print("10 + f1 = {}".format(50 + f1))
```

- What will Python do?

```
f1 + 10 = Instance of f, x = 60

Traceback (most recent call last):

  File "./foo.py", line 29, in <module>

    print("10 + f1 = {}".format(50 + f1))

TypeError: unsupported operand type(s) for
+: 'int' and 'Foo'
```

__radd__

- Auto-reversing: We get self and other, and now invoke our previously defined __add__:

```
def __radd__(self, other):

    return self.__add__(other)
```

How does this work?

- Python tries to do it the usual way
- It gets NotImplemented back
  - Not an exception!
  - An instance of Not ImplementedType!
- Python then tries (in desperation) to turn things around… and thus, __radd__

Type conversions

- You probably know about __str__ already
- But what about __int__? Or even __hex__?

Example

```
def __int__(self):
    return int(self.x)
def __hex__(self):
    return hex(int(self.x))
```

Boolean conversions

- Your object will always be considered True in a boolean context, unless you define __nonzero__ (__bool__ in Python 3)

Example

```
class Foo(object):
    pass
f = Foo()


>>> bool(f)
True
```

But with __nonzero__…

```
class Foo(object):
    def __init__(self, x):
        self.x = x
    def __nonzero__(self):
        return bool(self.x)

>>> f = Foo(1)
>>> bool(f)
True
>>> f = Foo(0)
>>> bool(f)
False
```

Format

- Originally, Python used the % syntax for pseudo-interpolation:

```
name = 'Reuven'

print('Hello %s' % name)
```

- But there are lots of problems with it, so it's better to use str.format

str.format

```
name = 'Reuven'

print('Hello {0}'.format(name))
```

Pad it!

```
name = 'Reuven'

print('Hello, {0:20}!'.format(name))
```

Move right

```
print('Hello, {0:>20}!'.format(name))
```

Custom formats

- It turns out that our objects can also handle these custom formats!
- If we define __format__ on our object, then it'll get the format code (i.e., whatever comes after the :)
- We can then decide what to do with it

```
class Person(object):
    def __init__(self, given, family):
        self.given = given
        self.family = family
    def __format__(self, format):
        if format == 'familyfirst':
            return "{} {}".format(self.family, self.given)
        elif format == 'givenfirst':
            return "{} {}".format(self.given, self.family)
        else:
            return "BAD FORMAT CODE"
```

Using it

```
>>> p = Person('Reuven', 'Lerner')

>>> "Hello, {}".format(p)

'Hello, BAD FORMAT CODE'


>>> "Hello, {:familyfirst}".format(p)

'Hello, Lerner Reuven'


>>> "Hello, {:givenfirst}".format(p)

'Hello, Reuven Lerner'
```

Pickle

- Pickle allows you to serialize, or marshall objects
- You can then store them to disk, or send them on the network
- Pickle works with most built-in Python data types, and classes built on those types

Pickling simple data

```
import pickle

d = {'a':1, 'b':2}

p = pickle.dumps(d)

new_d = pickle.loads(p)
```

Custom pickling

- Define some magic methods (of course) to customize your pickling:
- __getstate__ returns a dictionary that should reflect the object. Want to add to (or remove from) what is being pickled? Just modify a copy of self.__dict__ and return it!
- Don't modify the dictionary itself…

Example

```
class Foo(object):
    def __init__(self, x):
        self.x = x
    def __getstate__(self):
        odict = self.__dict__.copy()
        odict['y'] = self.x * 2
        return odict


>>> f = Foo(10)
>>> p = pickle.dumps(f)
>>> new_f = pickle.loads(p)
>>> vars(new_f)
{'x': 10, 'y': 20}
```

Equality

- What makes two object equal?
- By default in Python, they're only equal if their ids are equal
- (Yes, every object has a unique ID number — use the "id" function to find it!)
- You change this by defining __eq__!

Changing equality

```
class Foo(object):

    def __init__(self, x):

        self.x = x

    def __eq__(self, other):

       return self.x == other.x


>>> f1 = Foo(10)

>>> f2 = Foo(10)

>>> f1 == f2

True
```

Hashing

- If two objects are equal, then maybe they should hash to the same value, right?
- We can set the __hash__ attribute to be a method that returns whatever we want

Playing with __hash__

```
class Foo(object):
    def __init__(self, x):
        self.x = x
    def __hash__(self):
        return hash(self.x)
>>> f = Foo('a')
>>> hash(f)
12416037344
>>> hash('a')
12416037344
f = Foo(1)
>>> hash(f)
1
```

Attributes

- Objects in Python depend on having attributes
- Attributes are little namespaces associated with objects
- Each object has its own set of attributes

Listing attributes

- We can list attributes with dir() on an object
- This returns a list of strings
- That's nice, but how can we use these strings to understand the attributes?

getattr()

- Built-in function that takes two parameters — an object, and an attribute name

```
f.__class__

    __main__.Foo

getattr(f, '__class__')

    __main__.Foo
```

setattr()

- Not surprisingly, this built-in function sets an attribute's value

```
getattr(p, 'name')

    'Reuven'

setattr(p, 'name', 'Waldo')

getattr(p, 'name')

    'Waldo'

p.__dict__

    {'name': 'Waldo'}
```

What does this do?

```
def foo(): pass

[ name
  for name in dir(p)
  if type(getattr(p, name)) == type(foo) ]
```

__slots__

- Every instance has __dict__, a dictionary of name-value pairs defining its attributes
- This can waste a lot of memory if you're creating many instances with few attributes
- Solution: Define __slots__ as a class-level attribute

__slots__ example

```
class Person(object):
    __slots__ = ['first', 'last']
    def __init__(self, first, last):
        self.first = first
        self.last = last
p1 = Person('Reuven', 'Lerner')
p1.first
    'Reuven'
p1.last
    'Lerner'
p1.xxx = 'yyy'
    AttributeError
```

Properties

- Properties are "magic" attributes
- You define methods that should execute when the attribute is set or retrieved
- To the user, it appears as though they're just setting or getting an attribute — but really, anything could be happening

property()

- The old style of declaring properties worked as follows:
    - Define getter and setter methods, probably with _private names
    - Assign a class attribute to the return value of property(), which takes getter and setter methods as arguments

```
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def get_full_name(self):
        return "{} {}".format(self.first_name, self.last_name)
    def set_full_name(self, full_name):
        self.first_name, self.last_name = full_name.split()
    full_name = property(get_full_name, set_full_name)

p = Person('Reuven', 'Lerner')
print p.full_name        # prints "Reuven Lerner"
p.full_name = 'abc def'
print p.full_name        # prints "abc def"
```

@property

- Today, we can (and should) define properties with a decorator
- The syntax is a bit strange:
  - The getting function is decorated with @property
  - The setter function is decorated with @GETTERNAME.setter

```
#!/usr/bin/env python
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return "{} {}".format(self.first_name, self.last_name)

    @full_name.setter
    def full_name(self, full_name):
        self.first_name, self.last_name = full_name.split()

p = Person('Reuven', 'Lerner')
print p.full_name
p.full_name = 'abc def'
print p.full_name
```

When use properties?

- Type checking
- Value checking
- Black-boxing getter/setter functions
- Evolving your API from simple attributes to programmed ones

__getattr__

- For more flexibility on attribute access, you can define __getattr__
- This method is called when the attribute being retrieved does not exist
- If the attribute does exist, __getattr__ is not called

__getattr__ example

```
class Foo(object):
    def __init__(self):
        self.x = 100
    def __getattr__(self, attribute):
        return "{} doesn't exist.".format(attribute)

f = Foo()
print"f.x={}".format(f.x)       #100
print "f.y ={}".format(f.y)     # y doesn't exist
```

__setattr__

- Not surprisingly, there is also a __setattr__ method. (There is also a __delattr__)
- However, __setattr__, if defined, is called for all attributes, not just those that aren't found.
- If you want to set an attribute within __setattr__, you must explicitly call object.__setattr__. Otherwise, you get an infinite loop!

```
class Foo(object):
    def __init__(self):
        self.x = 100
    def __setattr__(self, attribute, value):
        print "Setting '{}' to '{}'".format(attribute, value)
        return object.__setattr__(self, attribute, value)

f = Foo()                        # Setting x to 100
print "f.x={}".format(f.x)       # 100

f.x = 200                        # Setting x to 200
print "f.x={}".format(f.x)       # 200
```

__getattribute__

- This is the same idea as __getattr__, except that it is invoked unconditionally
- __getattribute__ is called before Python checks to see if the attribute exists
- If you implement both, then only __getattribute__ will be invoked.

Descriptors

- You can think of descriptors as working like __getattribute__, but for a single attribute (rather than for all of them)
- You can also think of properties as simple descriptors
- Descriptors allow you to customize the retrieval, assignment, and deleting of the descriptor attribute
- They also let you reuse this functionality across many classes.

Definitions

- A descriptor is a Python class that implements one or more of the following methods:

```
__get__

__set__

__delete__ (not __del__ !)
```

When do we use them?

- Descriptor instances are used for instances of other classes' attributes
- So if you have a class with a particular attribute, you can set it to be an integer, string, list, or dict. Or an instance of any other object… including a descriptor instance.

In other words

- If we have a descriptor class named MyDescriptor, we can define a new class that uses it as follows:

```
class MyClass(object):

  a = MyDescriptor("a")

  b = MyDescriptor("b")
```

- Now a and b are class-level attributes of MyClass!

Remember

- Descriptors are defined at the class level
- They are class attributes, not instance attributes

Creating a descriptor

- Define a new class, inheriting (of course) from object
- In that class, define __init__, __get__, __set__, and/or delete
- On the class that will use the descriptor, assign a class-level attribute to an instance of the descriptor class

Basic descriptors

```
class Descriptor(object):
    def __init__(self):
        self._name = ''
    def __get__(self, instance, owner):
        print "Getting: {}".format(self._name)
        return self._name
    def __set__(self, instance, name):
        print "Setting: {}".format(name)
        self._name = name.title()
    def __delete__(self, instance):
        print "Deleting: {}".format(self._name)
        del self._name
class Person(object):
    name = Descriptor()
```

Using this code

```
user = Person()
user.name = 'john smith'
    Setting: john smith
user.name
    Getting: John Smith
    'John Smith'
del user.name
    Deleting: John Smith
```

Descriptor example #1

- Let's say we want to have Person.fullname (i.e., the user's first and last names)
- We could define a method, as we've done before
- But an attribute is more natural
- Solution: Define fullname as a descriptor

Descriptor class

```
class FullName(object):
    def __get__(self, instance, owner=None):
        return "{} {}".format(instance.first_name,
                              instance.last_name)
```

Things to notice

- __get__ not only gets its own instance (self), but the instance of the object on which it was invoked, and the class of that instance

- If all we want to do is compute a return value, we can merely implement __get__

- We're taking advantage of attribute scoping rules here

Why do this?

- In the case of FullName, we could just implement a method, right?
- Yes, definitely.
- But if you find yourself implementing the same things on many classes, a descriptor can help to DRY up your code

Descriptor example #2

- We can use descriptors to cache data
- Now we will use __init__ to set up our cache
- The first time it is used, our code will compute the result
- Subsequent calls will retrieve the cached data

```
class FullName(object):
    def __init__(self):
        self.cache = ''
    def __get__(self, instance, owner=None):
        if not self.cache:
            print "Caching!"
            self.cache = "{} {}".format(instance.first_name,
                                        instance.last_name)
        return self.cache

class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    fullname = FullName()

p1 = Person('Reuven', 'Lerner')
print p1.fullname      # Reuven Lerner
p2 = Person('Atara', 'Lerner-Friedman')
print p2.fullname      # Reuven Lerner -- uh oh!
```

What happened?

- Our caching worked, but only for the first instance!
- That's because the descriptor is instantiated only once per class! Storing the data in self.cache will only work once.
- Let's try a slightly different technique…

```
class FullName(object):
    def __init__(self):
        self.cache = { }
    def __get__(self, instance, owner=None):
        if id(instance) not in self.cache:
            print "Caching!"
            self.cache[id(instance)] = "{} {}".format(instance.first_name,
                                                      instance.last_name)
        return self.cache[id(instance)]

class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    fullname = FullName()

p1 = Person('Reuven', 'Lerner')
print p1.fullname             # Reuven Lerner

p2 = Person('Atara', 'Lerner-Friedman')
print p2.fullname             # Atara Lerner-Friedman
```

Multiple inheritance

- Python lets you inherit from multiple bases
- Just specify them in the class definition
- You then have access to methods defined in the base classes

```
class AccountHolder(object):
    def __init__(self, id_number, balance):
        self.id_number = id_number
        self.balance = balance
    def current_balance(self):
        return self.balance
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
    def fullname(self):
        return "{} {}".format(self.first_name,
                              self.last_name)
class CEO(Person, AccountHolder):
    def __init__(self, first_name,
                 last_name, id_number, balance):
        Person.__init__(self, first_name, last_name)
        AccountHolder.__init__(self, id_number, balance)
```

Notice __init__

- Kind of clunky, right?
- That's because we need to initialize our class with each of its bases
- One way to avoid some (but not all) of this mess is to use **kwargs as arguments — then we can pass a dict to all versions of __init__

Using our class

```
c = CEO('Reuven', 'Lerner', 123, 1000)
print "Current balance: {}".format(c.current_balance())
print "Full name: {}".format(c.fullname())
```

What about conflicts?

- What if multiple base classes define the same method name?
- Python uses a depth-first search to resolve the methods as best as possible
- As a general rule, earlier bases go first
- If you have a twisted inheritance scheme, consider using less inheritance

Mixins

- One use of multiple inheritance is mixing
- A mixin is a class designed to be used in multiple inheritance, and whose methods take precedence over other base classes' methods
- This lets you replace one or more methods of a class

Example

```
import locale
class CommafyMixin(object):
    def current_balance(self):
        locale.setlocale(locale.LC_ALL, 'en_US')
        return locale.format("%d", self.balance,
                                    grouping=True)
```

Using our mixin

- Now, for any class that has a current_balance() method, we can mix in our Commafy class:

```
class CEO(CommafyMixin, Person, AccountHolder):
    def __init__(self, first_name, last_name, id_number, balance):
        Person.__init__(self, first_name, last_name)
        AccountHolder.__init__(self, id_number, balance)
```

When to use mixing

- Only for methods, not for data!
- Provide many options features for a class via mixing
- Provide one feature in many different classes

type()

- Remember type?

```
a = 'abc'

type('') == type(a)

    True
```

type() also creates types!

```
X = type('X', (), {'cube':cube,
                    'square':square, 'y':5})
x = X()

x.cube(5)
      125

x.square(5)
      25
 x.y
      5
```

So what?

- This is interesting, but how does it affect me?
- Consider that every class is an object, and every object has a type that determines its template
- Maybe we want to create a different template for some of our objects?

Metaclasses

- Each class has a metaclass
- The metaclass determines the behavior of the class
- In most cases, you don't really want to be messing with these
- But even so, they're fun and interesting (if very confusing)

Slightly differently:

- Classes
  - are object factories
  - define the behavior of objects
  - define what it means to be an instance of the class

And:

- Metaclasses
  - are class factories
  - define the behavior of classes
  - define what it means to be a class

Default metaclass

- If all classes have a metaclass, then what is the default metaclass?
- That's right — the default metaclass is type
- And we learn the metaclass with type()

Changing the default

- You can assign a metaclass other than "type" by setting __metaclass__
- The value to which you assign __metaclass__ needs to be a metaclass

Creating a metaclass

- Just subclass "type" instead of "object"

```
class MetaClass(type): pass
```

Now we can use it!

```
class Foo(object):
  __metaclass__ = MetaClass
f = Foo()

type(f)
    <class '__main__.Foo'>
type(Foo)
    <class '__main__.MetaClass'>
type(MetaClass)
    <type 'type'>
```

Making our metaclass more interesting

- We can define __init__ on our metaclass
- This is what will be executed when a new class is created!
- That's where metaclasses shine, letting us execute code automatically when a class is *defined*, not just when it is instantiated.

metaclass __init__

- It receives four parameters (just as we passed to type):
  - cls — class
  - name — new class name (string)
  - bases — tuple of superclasses
  - dct — attributes to set

Metaclass definition

```
class CustomMetaclass(type):
    def __init__(cls, name, bases, dct):
        cls.x = 10
        print "Created {} using CustomMetaclass".format(name)
        super(CustomMetaclass, cls).__init__(name,
                                             bases, dct)
```

Using our metaclass

```
class BaseClass(object):
    __metaclass__ = CustomMetaclass
class Subclass1(BaseClass):
    pass
Creating class BaseClass using CustomMetaclass
Creating class Subclass1 using CustomMetaclass
```

Why metaclasses?

- Replace built-in behavior — for example, replace dump() with something else
- Set properties/attributes that will work on all classes
- Apply a decorator to some or all attributes on the new class
- Write a DSL (domain-specific language)

Why not metaclasses?

- They take a while to understand — generally considered "black magic"
- Most of what you want to do can probably be done more easily with decorators
- Hard to read and debug code written with them

But why metaclasses?

- They're cool and fun!
- They help you to understand the Python object system
- In some cases, they can reduce code and increase readability

Example: Permissions

```
import os
class PermissionMetaclass(type):
    def __init__(cls, name, bases, dct):
        print "Adding username"
        username = os.environ.get('MC_USERNAME', None)
        cls.username = username
        super(PermissionMetaclass, cls).__init__(name,
                                                 bases,
                                                 dct)
```

Using our metaclass

```
$ export MC_USERNAME=blah

class Foo(object):
    __metaclass__ = PermissionMetaclass
    def __init__(self):
        self.stuff = 'abc'

Adding username
Foo.username
    'blah'
```

What did we do?

- Every new instance of our metaclass will now have the "username" attribute
- But remember, we only have one chance to do something — at class-creation time!

Playing more

```
import os

class MyAbs(type):
    def __init__(cls, name, bases, dct):
        for key in dct:
            print "Class defines {}".format(key)
        super(MyAbs, cls).__init__(name, bases, dct)
```

Using this metaclass

```
>>> from myabs import *
>>> class Bar(object):
...
...
...
...
...
...
Class defines __module__
Class defines stuff
Class defines __metaclass__
Class defines __init__
```

Subclasses

```
>>> class Bar2(Bar):
     pass

Class defines __module__
```

Enforcement!

```
import os

class MyAbsError(RuntimeError):
    pass
```

```
class MyAbs(type):
    def __init__(cls, name, bases, dct):
        required_methods = ['a', 'b', 'c']

        failed_creation = False

        for method_name in required_methods:
            print "Checking for method '{}'".format(method_name)
            if not method_name in dct:
                print "No required method '{}'".format(method_name)
                failed_creation = True
                continue
            the_attribute = getattr(cls, method_name)
            if not callable(the_attribute):
                print "'{}' is not a method".format(method_name)
                failed_creation = True

        if failed_creation:
            raise MyAbsError, "No class for you..."
        super(MyAbs, cls).__init__(name, bases, dct)
```

Using MyAbs

```
print "\n\nCreating class 'Stuff'"
class Stuff(object):
    __metaclass__ = MyAbs
    def a(self): pass
    def b(self): pass
    def c(self): pass
print "\n\nCreating class 'Foo'"
class Foo(object):
    __metaclass__ = MyAbs
    def __init__(self):
self.x = 'x'
```

ABCMeta

- A metaclass that lets you ensure certain methods are defined in a class
- Included in the "abc" ("abstract base class") module, which comes with Python

Using ABCMeta

```
from abc import ABCMeta, abstractmethod
class Person(object):
  __metaclass__ = ABCMeta
  @abstractmethod
  def full_name(self): pass
```

Instantiate Person?

```
>>> p = Person()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class
Person with abstract methods full_name
```

Instantiate subclass?

```
class Employee(Person):
  pass

>>> e = Employee()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class
Employee with abstract methods full_name
```

But define full_name…

```
class Employee(Person):
  def full_name(self):
    return "Reuven Lerner"

>>> e = Employee()

>>> e.full_name()
'Reuven Lerner'
```

So remember:

- When we say:

```
class Name(object):
    x=5
```

- what is really going on?

What happens?

- The class body is executed as if it were a function
- The names in that function (variables + functions) are put into a dictionary
- We get the class's metaclass M

```
Name = M(Name, bases, d)
```

Metaclass limitations

- The metaclass doesn't enter the picture until after the class body executes!
  - We used that here to our advantage!
- Python 3 provides a metaclass keyword to class as a result

Do we need metaclasses?

- More than 50% of metaclass uses are now obviated by decorators (Alex M.)
- They can be useful for new types of behavior that cut across classes
- But again, your first instinct should be a decorator

__new__ vs. __init__

- Newcomers to Python are surprised to find that __init__ is an instance method
- Shouldn't a constructor be a class method?
- Yes, and that's why we have __new__.

__new__

- __new__ gets a type as its first parameter
- __new__ returns a new instance
- If it doesn't return a new instance, then __init__ isn't ever invoked

__init__

- Instance method
- Expects to get the instance in self
- Modifies the attributes of the instance
- Inappropriate for immutable types
  - So for those, use __new__ instead!

When to use __new__

- The big place to do it: When subclassing an immutable type
- Otherwise, think carefully before doing so