Classes and Object-Oriented Programming

CS 339R (Python) — Chapter 7

Spring 2011
Class Objects and Attributes

class Account(object):
    num_accounts = 0
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
        Account.num_accounts += 1
    def __del__(self):
        Account.num_accounts -= 1
    def deposit(self, amt):
        self.balance = self.balance + amt
    def withdraw(self, amt):
        self.balance = self.balance - amt
    def inquiry(self):
        return self.balance

As usual, all attribute access must be qualified.
# Create a few accounts
a = Account("Guido", 1000.00)  # Account.__init__(a,"Guido", 1000.00)
b = Account("Bill", 10.00)

a.deposit(100.00)        # Calls Account.deposit(a,100.00)
b.withdraw(50.00)        # Calls Account.withdraw(b,50.00)
name = a.name            # Get account name
class Foo(object):
    def bar(self):
        print("bar!")
    def spam(self):
        bar(self)  # Incorrect! 'bar' generates a NameError
        self.bar() # This works
        Foo.bar(self) # This also works

The first call to **bar** in **spam** follows the LEGB rule for name lookup
Inheritance

• Syntax:
  • `class myclass(base1, base2, ...): <body>`

• Inheritance in Python is mostly just a name lookup specification:
  • *Lookup order*: object, class, superclasses (left-to-right, bottom-to-top — sort of)
  • Overridden methods are simply found first (i.e., polymorphism is *automatic*)

• Data is not “inherited” as in other languages
  • Because everything is *dynamically bound*
  • Calling superclass methods is how “inherited data” is bound to objects
    • `self` is passed, as usual
class EvilAccount(Account):
    def __init__(self, name, balance, evilfactor):
        Account.__init__(self, name, balance) # Initialize acc't
        self.evilkfactor = evilfactor

    def inquiry(self):
        if random.randint(0, 4) == 1:
            return self.balance * self.evilkfactor
        else:
            return self.balance

A superclass constructor is *overridden* by a local __init__(like any other method)
Calling Methods “Way Up There”

- Must be qualified with the class name
- If you want search beyond the current class, call super
  - super: “start above the given class in the hierarchy and look for the name”
  - For providing “before/after” logic around an inherited method
  - See next slide...
Using `super(cls, obj)`

# NOTE: deposit is in the Account class only

class MoreEvilAccount(EvilAccount):
    def deposit(self, amount):
        self.withdraw(5.00)  # Subtract the "convenience" fee
        EvilAccount.deposit(self, amount)  # Now, make deposit

# The following is identical:
class MoreEvilAccount(EvilAccount):
    def deposit(self, amount):
        self.withdraw(5.00)  # Subtract "convenience" fee
        super(self.__class__, self).deposit(amount)  # Make deposit

In Python 3 you only need to say `super().deposit(amount)`
See also: `super.py`
Multiple Inheritance

* If an attribute occurs in more than one parent class, which one do you use?

* “To find attributes with multiple inheritance, all base classes are ordered in a list from the ‘most specialized’ class to the ‘least specialized class’. Then, when searching for an attribute, this list is searched in order until the first definition of the attribute is found.”

* most specialized: earlier in the list of base classes, lower in the hierarchy

* class MostEvilAccount(EvilAccount, DepositCharge, WithdrawCharge)
Method Resolution Order (MRO)

- aka “Linearization”
- Obvious in linear hierarchies
- A little more interesting with multiple inheritance!

Given class $C(A_1, A_2, ..., A_n)$: $L(C) = C + \text{merge}(L(A_1), ..., L(A_n))$

$L(\text{object}) = \text{object}$  \# $\text{merge}$: keeps last occurrence only

Consider class $A(\text{object})$; class $B(\text{object})$; class $C(A, B)$;
Given class \( C(A_1,A_2,\ldots,A_n) \): \( L(C) = C + \text{merge}(L(A_1),\ldots,L(A_n)) \)

\( L(\text{object}) = \text{object} \quad \# \text{ merge: keeps last occurrence only} \)

class A(object): pass
class B(object): pass
class C(A, B): pass
print C.__mro__

\[
\begin{align*}
L(A) &= A + \text{merge}(L(\text{object})) = [A, \text{object}] \\
L(B) &= B + \text{merge}(L(\text{object})) = [B, \text{object}] \\
L(C) &= C + \text{merge}(L(A), L(B)) \\
&= C + \text{merge}([A, \text{object}], [B, \text{object}]) \\
&= C + [A, B, \text{object}] \\
&= [C, A, B, \text{object}]
\end{align*}
\]
class Animal(object):
    def __init__(self, name):
        print 'Animal.__init__'
        self.name = name
    def whoAmI(self):
        return self.name

class Dog(Animal):
    def __init__(self, name):
        print 'Dog.__init__'
        super(Dog, self).__init__(name)
    def speak(self):
        print "Bark!"

class Antelope(Animal):
    def __init__(self, name):
        print 'Antelope.__init__'
        super(Antelope, self).__init__(name)
    def speak(self):
        print "<silent>"

class Basselope(Dog, Antelope):
    def __init__(self, name):
        print 'Basselope.__init__'
        super(Basselope, self).__init__(name)

bl = Basselope("Rosebud")
print bl.whoAmI(),'\n',
bl.speak()
print Basselope.__mro__
What is the MRO for E,F,G?
An Illegal Hierarchy

Y follows X in Z’s specification, but is more specific than X ⇒ conflict!
(C++ uses Z->Y->X)
Multiple Inheritance Rule

- Rule: “If C1 precedes C2 in the linearization of C, then C1 precedes C2 in the linearization of any subclass of C. Otherwise, the innocuous operation of deriving a new class could change the resolution order of methods, potentially introducing very subtle bugs.”
Adding Methods Dynamically

def eat(x, food):
    # x will be self
    print x.whoAmI(), 'eating', food

Dog.eat = eat
    # Add new method to Dog!

muffy = Dog('Muffy')
muffy.eat('trash')
    # muffy becomes x in Dog.eat()
Dog.eat(muffy, 'bones')

# Output:
Muffy eating trash
Muffy eating bones
Adding an Object-only “Method”

- Not commonly done
- Must pass “self” explicitly
- Only true methods belonging to a class can be called as x.f()

```python
>>> muffy = Dog('Muffy')
>>> def eat(x, food):
...     print x.whoAmI(), 'prefers to eat socks instead of', food
...     print ...

>>> muffy.f = eat
>>> muffy.f(muffy, 'trash')
Muffy prefers to eat socks instead of trash
```
Unbound Methods

- If method is non-static, an object must be supplied later

```python
op = Dog.whoAmI  # no self
print op
print op(muffy)  # same as muffy.whoAmI()

# Output:
<unbound method Dog.whoAmI>
Muffy
```
Bound Methods

```
sheba = Dog('Sheba')
op = sheba.whoAmI  # self is sheba's object
print op
print op()        # same as sheba.whoAmI()
map(muffy.eat, ['melon', 'bones'])
# == [muffy.eat('melon'), muffy.eat('bones')]

# Output:
<bound method Dog.whoAmI of <__main__.Dog object at 0x009FF130>>
Sheba
Muffy eating melon
Muffy eating bones
```
Static Methods Redux

- Static methods are *normal functions*
- Except they are *attributes* of a class
- They receive no `self` parameter
- Call with `classobj.methodname(...)`
  - Although can also call with `obj.methodname(...)`
- See `static.py`
Class Methods

* Similar to static methods, but...

* They implicitly receive the *class object* involved in the original call

* See *date.py*
Counting Subclass Instances

- An application of class methods
- Need a separate counter for each subclass
- So, put it in the unique subclass object, but automatically
- See counted2.py
Bound Class Methods

# Class Methods are Bound Methods
# (Bound to their class object, of course)

m = Line.showCount
print m
m()  # same as Shape.showCount(Line)

# Output:
<bound method type.showCount of <class '__main__.Line'>>>
Class Line has count: 1
Properties

- Allows calling methods without explicit function-call syntax
- Looks like you’re accessing a data attribute
  - But a function is actually called
  - Suitable for managed and computed attributes
- Use the @property decorator:
  - You can provide a getter (default), setter, and deleter
  - If omitted, the operation is not available
  - So only providing a getter makes an attribute read-only
Property Syntax

class C(object):
    def __init__(self):
        self.__x = None

@property  # The getter
def x(self):
    """I'm the 'x' property."""
    return self.__x

@x.setter
def x(self, value):
    self.__x = value

@x.deleter
def x(self):
    del self.__x

Sample Execution:

c = C()
c.x = 10  # calls setter
print vars(c)
print c.x  # calls getter
print C.x.__doc__
del c.x    # calls deleter
print vars(c)

''' Output:
{'_C__x': 10}
10
I'm the 'x' property.
{}''
Mangled Attribute Names

* Python offers *no private access specification*
  
* Instead, use *two leading underscores*
  
  * The actual attribute will start with ‘_classname’*
  
  * e.g., C.__foo ==> C._C__foo*

* Recommended for *mutable* attributes
  
  * And use *properties* to manage them*
Object Serialization

- A technique for object persistence
  - They get stored in files
  - You can retrieve them later
  - All attributes are properly restored

- The **pickle** module:
  - `pickle.dump(x, <file>)`
  - `x = pickle.load(<file>)`
  - See next slide
Using Pickle

```python
>>> f = open('Dog.dat', 'w')
>>> pickle.dump(muffy, f)
>>> pickle.dump(sheba, f)
>>> f.close()
>>> del muffy
>>> del sheba
>>> f = open('Dog.dat')
>>> muffy = pickle.load(f)
>>> sheba = pickle.load(f)
>>> f.close()
>>> muffy
<animal.Dog object at 0x100498a50>
>>> sheba
<animal.Dog object at 0x100500910>
>>> muffy.whoAmI()
'Muffy'
```
Operator Overloading

- Done by defining *special methods*
  - `__call__`, `__add__`, etc.
- See the `operator` module for special names for operators
class Number(object):
    def __init__(self, num):
        float(num)  # correctness test
        self.__num = num
    def __add__(self, num):
        if (isinstance(num, Number)):
            return Number(self.__num + num.__num)
        return Number(self.__num + float(num))
    def __str__(self):
        return '%g' % self.__num
    def __neg__(self):
        return Number(-self.__num)
__radd__ = __add__    # Adds the __radd__ attribute
x = Number(1.2)
y = Number(3.4)
print x,y,-x
print x + y
print x + 2
print 2 + x  # calls __radd__(x,2) == __add__(x,2)

#Output:
1.2 3.4 -1.2
4.6
3.2
3.2
Indexing Your Objects

- The method `__getitem__(self, i)` is called when fetching `self[i]`
- Likewise, `__setitem__(self, i, x)` processes `self[i] = x`
- Iterator contexts automatically call these functions
  - If no other iterator is provided
  - They call `__getitem__` until a `StopIteration` is raised
- See next two slides
class Stuff(object):
    def __init__(self):
        self.__data = []
    def add(self, x):
        self.__data.append(x)
    def __getitem__(self, i):
        return self.__data[i]
    def __setitem__(self, i, x):
        self.__data[i] = x
    def display(self):
        for item in self.__data:
            print item,
            print
    def __len__(self):
        return len(self.__data)
Indexing Example

```python
s = Stuff()
s.add(2)
s.add('three')
s.add(4.0)
s.display()
for i in range(len(s)):
    print s[i],
print
s[1] = 'one'
s.display()

# Output:
2 three 4.0
2 three 4.0
2 one 4.0
```
You can define your own iterators explicitly
- They can be used for loops and in other iterable contexts
- Loops call `iter()` to get an iterator

The `iter()` built-in will in turn try your `__iter__`, or `__getitem__` (in that order)
- Iteration is automatic if you define `__getitem__`!
Iteration via \texttt{\_\_getitem\_\_}

# Still using Stuff from 2 slides back
for x in s:
    print x,
print
print 'one' in s
print map(None, s)
print list(s)
print tuple(s)

# Output:
2 one 4.0
True
[2, 'one', 4.0]
[2, 'one', 4.0]
(2, 'one', 4.0)
Using `__iter__`

- Better for *one-pass traversal* than `__getitem__`
  - You must implement a `next` method and maintain state
  - Faster than `__getitem__`, since state is maintained across calls
  - See next two slides
Using `__iter__` and `next`

```python
class MoreStuff(object):
    def __init__(self):
        self.__data = []
    def add(self, x):
        self.__data.append(x)
    def __iter__(self):
        self.__pos = 0  # Initialize iterator state
        return self
    def next(self):
        if self.__pos == len(self.__data):
            raise StopIteration
        val = self.__data[self.__pos]
        self.__pos += 1
        return val
    def display(self):
        for item in self.__data:
            print item,
        print
```
Sample Execution

```python
s = MoreStuff()
s.add(2)
s.add('three')
s.add(4.0)
s.display()
for x in s:    # calls __iter__
    print x,
print
# Output:
2 three 4.0
2 three 4.0
```
The Object Instantiation Process

x = MyClass(args)

# Is Equivalent to:
x = MyClass.__new__(MyClass, args)  # calls object.__new__
MyClass.__init__(x, args)

- You can provide a custom __new__ for your class
- See new.py
Classes are Objects too

- Because *everything* is
- For every class there is *one, unique class object*
- This object is instantiated when Python processes it *class* statement
- The question is...
What is a Class Object’s Type?

* Every object has a type, class objects included
* When you say `foo = Foo()`, `Foo` instantiates the object `foo`
* What type instantiates the class object `Foo`? What is its `type`?
The type type

- By default, all types (classes) are instances of a special type, `type`
- `type` instantiates all class objects (unless otherwise specified by you)
- This is not an inheritance relationship!
- A class’s type is called its `metaclass`
The **type** Metaclass

```python
>>> class C(object) : pass
>>> c = C()
>>> type(c)
<class '__main__.C'>
>>> type(C)
<type 'type'>

>>> type(1)
<type 'int'>
>>> type(int)
<type 'type'>

>>> type(type)
<type 'type'>
```
Another Metaclass Perspective

* Classes determine the behavior of all of their object instances

* Metaclasses determine the behavior of all of their class instances

```python
dir(type)
['__abstractmethods__', '__base__', '__bases__', '__basicsize__', '__call__', '__class__', '__delattr__', '__dict__', '__dictoffset__', '__doc__', '__eq__', '__flags__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__instancecheck__', '__itemsize__', '__le__', '__lt__', '__module__', '__mro__', '__name__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasscheck__', '__subclasses__', '__subclasshook__', '__weakrefoffset__', 'mro']
```
The *Class Object* Instantiation Process

```python
class MyClass(bases): ...

# Is Equivalent to:
1) Execute the statements in *MyClass*
2) Store the resulting attributes in a new dictionary, *d*
3) Determine the metaclass, *mcs* (*type* by default)

# The rest is similar to object instantiation

MyClass = mcs.__new__(mcs,'MyClass',bases,d)
mcs.__init__(MyClass,'MyClass',bases,d)
```

* See *metanew.py*
A Custom Metaclass

# meta.py
class MyMetaClass(type):  # Derive from type
    def __str__(cls): return 'Class ' + cls.__name__

class C(object):
    __metaclass__ = MyMetaClass  # Assign metaclass (Python 2.x)

x = C()
print type(x)

# Output:
Class C

If no __metaclass__ is supplied inside a class, the metaclass of the first base class (object, here) is used. It is always type unless otherwise defined as above.
**Requiring a Method**

- This metaclass requires its classes to have a \texttt{__str__} method

```python
# metastr.py: A metaclass that forces a \texttt{__str__} method

class StrMeta(type):
    def __init__(cls, name, bases, d):
        if '__str__' not in d:
            raise TypeError('Missing __str__ method')
        type.__init__(cls, name, bases, d)

class C1(object):
    __metaclass__ = StrMeta
    def __str__(self):
        return 'StrMeta'

class C2(object):
    __metaclass__ = StrMeta

Traceback (most recent call last):
  File "metastr.py", line 14, in <module>
    class C2(object):
  File "metastr.py", line 6, in __init__
    raise TypeError('Missing __str__ method')
TypeError: Missing __str__ method
```
The \_\_slots\_\_ Attribute

- Limits the attributes an object can have
- Uncomment line 2 below:

```python
# slots.py
class C(object):
    # __slots__ = ['foo', 'bar']
    def __init__(self):
        self.foo = 0
        self.bar = 1

c = C()
c.x = 2
print c.foo, c.bar, c.x  # 0 1 2
```
Adding Getters Automatically

```python
# A metaclass that provides getters for all slots:
class Getters(type):
    @staticmethod
    def __new__(mcs, name, bases, d):
        for var in d.get('__slots__'):
            d['get' + var] = lambda self, v=var: getattr(self, v)
        return type.__new__(mcs, name, bases, d)

class G(object):
    def __init__(self, f, b):
        self.foo = f
        self.bar = b
    __metaclass__ = Getters
    __slots__ = ['foo', 'bar']

g = G(1,2)
print g.getfoo(), g.getbar()  # 1 2
```
Other Common Uses for MetaClasses

- Logging
- Interface checking
- Automatic property creation
- Automatic resource management
Abstract Base Classes

* For defining explicit *interfaces*
  * Client classes inherit from and *implement* them
  * May contain implementation for subclasses to share
* ABCs can’t be instantiated
  * Accomplished by using a *special metaclass*: `ABCMeta`
    * `from abc import ABCMeta, abstractmethod, abstractproperty`
* See pp. 136–137
from abc import ABCMeta, abstractmethod

class foo(object):
    __metaclass__ = ABCMeta  # special metaclass  
    @abstractmethod
    def bar(self): pass
    @abstractmethod
    def baz(self): pass

class spam(foo):
    def bar(self): pass
    def baz(self): pass

f = spam()