61A LECTURE 12 – OOP 2, INHERITANCE

Steven Tang and Eric Tzeng
July 14, 2013

Announcements

- Midterm grades are up
  - Average: 34
  - First quartile: 26.5
  - Median: 36.0
  - Third quartile: 43.0
- Hog contest strategy due today!

Review time

- You've seen all this before, so we're going to try to go a little faster…
- …but it was the day of the midterm, so we understand if the stuff is a little hazy
- Ask questions/slow me down if necessary!

Recall: Objects

- Everything in Python is an object
- Every object has a "type"
- An object's type (essentially, its "class") determines the set of behaviors and attributes that each object has

```
>>> x = 4
>>> y = 5
>>> x.real
4
>>> y.real
5
```

```
>>> s = [9, 5, 12, 7]
>>> s.sort
<built-in method sort ...>
>>> s.sort()
>>> s
[5, 7, 9, 12]
```

- `x and y are both int type: both have a real component, but different local values`

Object-Oriented Programming

A method for organizing modular programs

- Abstraction barriers
- Message passing
- Bundling together information and related behavior

A metaphor for computation using distributed state

- Each *object* has its own local state.
- Each object also knows how to manage its own local state, based on the messages it receives.
- Several objects may all be instances of a common type.
- Different types may relate to each other as well.

Specialized syntax & vocabulary to support this metaphor

Classes

A *class* serves as a template for its *instances*.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

Idea: All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

Better idea: All bank accounts share a "withdraw" method.

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

The Class Statement

```
class <name>(<base class>):
    <suite>
```

Discussed later

A class statement **creates** a new class and **binds** that class to <name> in the first frame of the current environment.

Statements in the <suite> create attributes of the class.

As soon as an instance is created, it is passed to __init__, which is an attribute of the class.

```
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Initialization

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes.

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:
2. The constructor __init__ of the class is called with the new object as its first argument (called self), along with additional arguments provided in the call expression.

```
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('Jim')
>>> b = Account('Jim')
```

Identity testing is performed by "is" and "is not" operators:

```
>>> a is b
False
>>> a is not b
True
```

Binding an object to a new name using assignment **does not** create a new object:

```
>>> c = a
>>> c is a
True
```

Methods

Methods are defined in the suite of a class statement

```
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

These def statements create function objects as always, but their names are bound as attributes of the class.

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

```
class Account(object):
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Called with two arguments

Dot notation automatically supplies the first argument to a method.

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Invoked with one argument

Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

<expression> . <name>

The <expression> can be any valid Python expression

The <name> must be a simple name

Evaluates to the value of the attribute **looked up** by <name> in the object that is the value of the <expression>

```
tom_account.deposit(10)
```

Dot expression

Call expression

Accessing Attributes

Using **`getattr`**, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:

- One of its instance attributes, **or**
- One of the attributes of its class

Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and
- *Bound methods*, which couple together a function and the object on which that method will be invoked.

```
Object  +  Function  =  Bound Method
```

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom_account.deposit)
<class 'method'>

>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1000)
2011
```

Attributes, Functions, and Methods

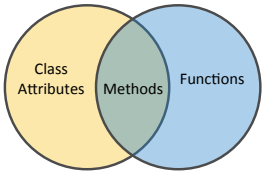
All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attributes: attributes of instance objects

Class attributes: attributes of class objects

Terminology:



Python object system:

Functions are objects.

Bound methods are also objects: a function that has its first parameter "self" already bound to an instance.

Dot expressions on instances evaluate to bound methods for class attributes that are functions.

Looking Up Attributes by Name

`<expression> . <name>`

To evaluate a dot expression:

1. Evaluate the <expression>.
2. <name> is matched against the instance attributes.
3. If not found, <name> is looked up in the class.
4. That class attribute value is returned unless it is a **function**, in which case a *bound method* is returned.

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance.

```
class Account(object):

    interest = 0.02         # Class attribute

    def __init__(self, account_holder):
        self.balance = 0    # Instance attribute
        self.holder = account_holder

    # Additional methods would be defined here
```

```
>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```

`interest` is not part of the instance that was somehow copied from the class!

Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

Instance Attribute Assignment:

```
tom_account.interest = 0.08
```

This expression evaluates to an object

But the name ("interest") is not looked up

Attribute assignment statement adds or modifies the "interest" attribute of tom_account

Class Attribute Assignment:

```
Account.interest = 0.04
```

Practice

- Make a `Dog` class
- To create a `Dog` instance, provide a `name` that will be kept track of
- Dogs keep track of their `hunger`, which starts at 0
- You can ask Dogs to `speak()`
  - Doing so increases their `hunger` by 1 and returns `'woof'`
- You can have a `Dog eat()`
  - This decreases `hunger` by 1

```
>>> beagle = Dog('snoopy')
>>> snoopy.name
'snoopy'
>>> snoopy.speak()
'woof'
>>> snoopy.speak()
'woof'
>>> snoopy.hunger
2
```

Break!

Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization

Two classes have overlapping attribute sets, but one represents a special case of the other.

```
class <name>(<base class>):
    <suite>
```

Conceptually, the new subclass "shares" attributes with its base class.

The subclass may override certain inherited attributes.

Using inheritance, we implement a subclass by specifying its difference from the base class.

Inheritance Example

A **`CheckingAccount`** is a specialized type of **`Account`**.

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class **`Account`**

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + self.withdraw_fee)
```

Looking Up Attribute Names on Classes

Base class attributes *aren't copied* into subclasses!

To look up a name in a class.

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```

Designing for Inheritance

Don't repeat yourself; use existing implementations.

Attributes that have been overridden are still accessible via class objects.

Look up attributes on instances whenever possible.

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + self.withdraw_fee)
```

Attribute look-up on base class

Preferable alternative to `CheckingAccount.withdraw_fee`

General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same **`CheckingAccount`** behavior; different approach

```
class Account(object):
    interest = 0.02
    withdraw_fee = 0
    def withdraw(self, amount):
        amount += self.withdraw_fee
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

class CheckingAccount(Account):
    interest = 0.01
    withdraw_fee = 1
```

May be overridden by subclasses

Nothing else needed in this class

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor.

Inheritance is best for representing is-a relationships.

E.g., a checking account is a specific type of account.

So, **`CheckingAccount`** inherits from **`Account`**.

Composition is best for representing has-a relationships.

E.g., a bank has a collection of bank accounts it manages.

So, A bank has a list of **`Account`** instances as an attribute.

No local state at all? Just write a pure function!

More practice!

- Write a `Collie` class that does pretty much the same thing as the `Dog` class…
- Except when you tell it to `speak()`, it returns `'there is a boy trapped in the well'` instead of `'woof'`
- And when you tell it to `eat()`, it returns `'this food is exquisite'` instead of `None`