61A Lecture 17

Friday, October 5

Implementing an Object System

Today's topics:

- What is a class?
- What is an instance?
- How do we create inheritance relationships?
- How do we write code for attribute look-up procedures?

Tools we'll use:

- Dispatch dictionaries
- Higher-order functions

The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:

- Objects with **local state** & interact via **message passing**
- Objects are **instantiated** by classes, which are also objects
- Classes may **inherit** from other classes to share behavior
- Mechanics of objects are governed by **"evaluation procedures"**

THE LINE

Below the Line:

- Objects have **mutable dictionaries** of attributes
- **Attribute look-up for instances** is a function
- **Attribute look-up for classes** is another function
- Object **instantiation** is another function

Implementing the Object Abstraction

Fundamental OOP concepts:

- Object instantiation and initialization
- Attribute look-up and assignment
- Method invocation
- Inheritance

Not-so-fundamental issues (that we'll skip):

- Dot expression syntax
- Multiple inheritance
- Introspection (e.g., what class does this object have?)

Dot expressions are equivalent to getattr and setattr (Demo)

Instances

Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"

```
def make_instance(cls):
    """Return a new object instance."""

    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)

    def set_value(name, value):
        attributes[name] = value

    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
```

The class of the instance

Match name against instance attributes

Look up the name in the class

Assignment always creates/modifies instance attributes

Bound Methods

If looking up a name returns a class attribute value that is a function, getattr returns a bound method

```
def make_instance(cls):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)
    ...


def bind_method(value, instance):
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

Classes

Dispatch dictionaries with messages 'get', 'set', and 'new'

```
def make_class(attributes={}, base_class=None):
    """Return a new class."""

    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)

    def set_value(name, value):
        attributes[name] = value

    def new(*args):
        return init_instance(cls, *args)

    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
```

The class attribute look-up procedure

Common dispatch dictionary pattern

Instantiation and Initialization

First makes a new instance, then invokes the __init__ method

```
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...


def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args."""
    instance = make_instance(cls)
    init = cls['get']('__init__')
    if init:
        init(instance, *args)
    return instance
```

Dispatch dictionary

The constructor name is fixed here

Example: Defining an Account Class

```
def make_account_class():

    interest = 0.02

    def __init__(self, account_holder):
        self['set']('holder', account_holder)
        self['set']('balance', 0)

    def deposit(self, amount):
        new_balance = self['get']('balance') + amount
        self['set']('balance', new_balance)
        return self['get']('balance')

    def withdraw(self, amount):
        balance = self['get']('balance')
        if amount > balance:
            return 'Insufficient funds'
        self['set']('balance', balance - amount)
        return self['get']('balance')

    return make_class(locals())

Account = make_account_class()
```

Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['get']('holder')
'Jim'
>>> jim_acct['get']('interest')
0.02
>>> jim_acct['get']('deposit')(20)
20
>>> jim_acct['get']('withdraw')(5)
15
```

How can we also use getattr and setattr style syntax?

Class and Instance Attributes

Instance attributes and class attributes can share names

```
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['set']('interest', 0.08)
>>> Account['get']('interest')
0.02
```

Demo

Example: Using Inheritance

CheckingAccount is a special case of Account

```
def make_checking_account_class():

    interest = 0.01
    withdraw_fee = 1

    def withdraw(self, amount):
        fee = self['get']('withdraw_fee')
        return Account['get']('withdraw')(self, amount + fee)

    return make_class(locals(), Account)

CheckingAccount = make_checking_account_class()
```

Demo

Relationship to the Python Object System

Bonus Material

Object attributes are stored as dictionaries

Some "magic" names, __<name>__, require special handling

An object has an "attribute" called __dict__ that is a dictionary of its instance attributes

Demo

In Python, classes have classes too

The equivalent of init_instance can be customized (metaclass)