61A LECTURE 11 – OOP

Steven Tang and Eric Tzeng

July 11, 2013

Announcements

- Midterm! Don’t stress too much.
  - 7pm
  - 2050 VLSB for logins aa-hz
  - 10 Evans for logins ia-zz
- Hog contest strategy due Monday!

Where are we?

- Weeks 1 and 2:
  - The power of **functions** and **functional programming**
    - Can perform useful computations, like Newton's Method and Count Change
    - Can simulate games, like Hog
    - Utilize **data abstraction** to deal with complex programs
    - Can use **recursion** to express and solve certain types of problems
- Week 3:
  - What about other interesting problems, like modelling things that change?
  - A lead-up to **Object Oriented Programming**
    - Instead of creating a new function to do everything, let's bundle data and behavior together, and have each object perform computation
    - An extremely powerful metaphor that allows coding to be efficient and simple
    - Heavily relies on **mutating** the environment to update information

The Story So Far About Data

Data abstraction: Enforce a separation between how data values are represented and how they are used.

Abstract data types: A representation of a data type is valid if it satisfies certain behavior conditions.

Message passing: We can organize large programs by building components that relate to each other by passing messages.

Dispatch functions/dictionaries: A single object can include many different (but related) behaviors that all manipulate the same local state.

(All of these techniques can be implemented using only functions and assignment.)

A Mutable Container

```
def container(contents):
    """Return a container that is manipulated by two
    functions.

    >>> get, put = container('hello')
    >>> get()
    'hello'
    >>> put('world')
    >>> get()
    'world'
    """

    def get():
        return contents

    def put(value):
        nonlocal contents
        contents = value

    return put, get
```

Two separate functions to manage! Can we make this easier?

Dispatch Functions

A technique for packing multiple behaviors into one function

```
def pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:

- One conditional statement with several clauses
- Headers perform equality tests on the message

An Account as a Dispatch Dictionary

```
def account(balance):
    """Return an account that is represented as a
    dispatch dictionary."""

    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']

    def deposit(amount):
        dispatch['balance'] += amount
        return dispatch['balance']

    dispatch = {'balance': balance, 'withdraw': withdraw,
                'deposit': deposit}

    return dispatch
```

Question: Why dispatch['balance'] and not balance?

Object Oriented Programming

- Message passing seems like a good idea
  - Data can respond to lots of different requests - we can have powerful **data**
- Mutable local state seems like a good idea
  - Humans relate to this – things change in real life all the time
- Let’s program using both of these ideas. Python provides us with convenient OOP syntax
- Warning: Lots of new syntax! Best learning occurs through hands-on practice. Be sure to go to lab next week.

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

Interpreter session

- Recall the account abstraction created with dispatch dictionaries:

```
def account(balance):
    def withdraw(amount):
        ...

    def deposit(amount):
        ...

    dispatch = {'balance': balance, 'withdraw': withdraw,
                'deposit': deposit}

    return dispatch
```

- Let’s create a similar account, except let’s use Python’s object notation

Classes and Objects

- Every object is an instance of some particular class – use “type(obj)” to find which class
- The objects we have used so far in the course have all been created from built-in Python classes, but we can create our own
- Creating a new class is essentially making a new abstract data type. Inside the class definition, all of the objects’ behavior is specified.

A class is a blueprint of behaviors for creating objects

Every object created from that blueprint will have that certain set of behaviors

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
```

Break

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
```

Methods and Currying

Earlier, we saw *currying*, which converts a function that takes in multiple arguments into multiple chained functions.

The same procedure can be used to create a bound method from a function

```
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer
```

```
>>> add2 = curry(add)(2)
>>> add2(3)
5

>>> tom_deposit = curry(Account.deposit)(tom_account)
>>> tom_deposit(1000)
3011
```

Attributes, Functions, and Methods

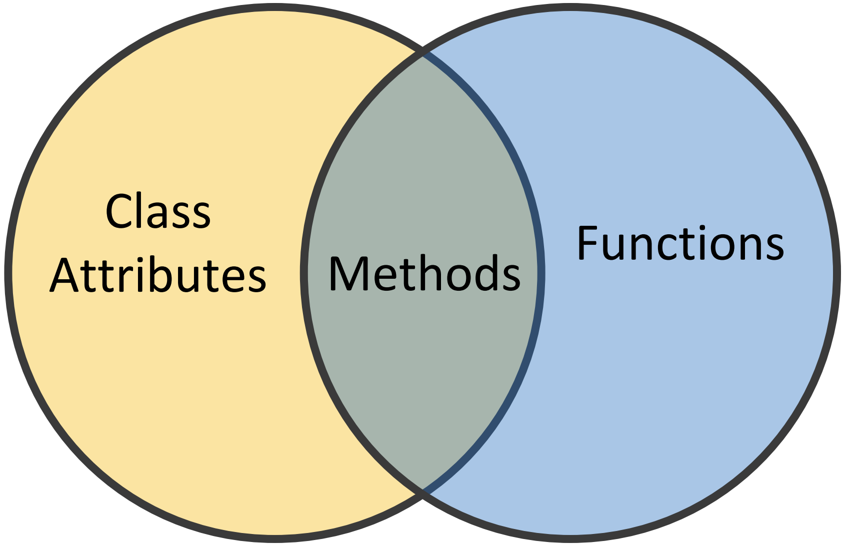
All objects have attributes, which are name-value pairs

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
```

Attribute Assignment Statements

Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

```
balance:   0
holder:    'Jim'
interest:  0.08
```

```
balance:   0
holder:    'Tom'
```

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

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