61A Lecture 15

Monday, October 1

Dispatch Dictionaries

A dispatch dictionary has messages as keys and functions (or data objects) as values.

Dictionaries handle the message look-up logic; we concentrate on implementing useful behavior.

Demo

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.)

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

Next lecture

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

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('Jack')
```

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

```
>>> a is a
True
>>> 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

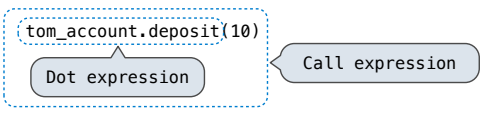
Dot notation accesses attributes of the instance **or** its class

<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>



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
```


Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression.
2. <name> is matched against the instance attributes of that object; **if an attribute with that name exists**, its value is returned.
3. If not, <name> is looked up in the class, which yields a class attribute value.
4. That value is returned **unless it is a function**, in which case a *bound method* is returned instead.

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   # A class attribute

    def __init__(self, account_holder):
        self.balance = 0
        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 Statements and 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

```
>>> 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
```