CS61A Lecture 20
Object-Oriented Programming: Implementation
Jom Magrother
UC Berkeley EECS
July 23, 2012

COMPUTER SCIENCE IN THE NEWS
Researchers create memory with one bit per molecule
July 19, 2012

TODAY
- Review: Dispatch Functions
- Dispatch Dictionaries
- OOP Implementation

REVIEW: DISPATCH FUNCTIONS
We saw that we do not need data structures to store information: we can use functions to accomplish the storage and retrieval of data.

```
def make_pair(first, second):
    def pair(msg, arg=None):
        nonlocal first, second
        if msg == 'first':
            return first
        elif msg == 'second':
            return second
        elif msg == 'set-first':
            first = arg
        elif msg == 'set-second':
            second = arg
        else:
            return "Message not understood."
    return pair
```
**Review: Dispatch Functions**

The inner function `pair`, returned by every call to `make-pair`, represents the “pair” object. It receives a `message` as an argument, and responds by dispatching the appropriate piece of code.

The function `pair` is called the **dispatch function**; this programming style is called **message passing**.

**Message Passing**

Enumerating different messages in a conditional statement is not very convenient.

- Equality tests can be repetitive.
- We need to write new code for new messages.

Can we use something that will take care of the message lookup and code dispatch for us?

**Dispatch Dictionaries**

**Idea:** We will allow ourselves one kind of data structure. In particular, we will represent an object by a **dispatch dictionary**, where the `messages` are the keys.

**Dispatch Dictionaries**

```python
def make_pair(first, second):
    pair_dict = {'first': first, 'second': second}
    return pair_dict
```

How do we create and use “pair objects” now?

```python
>>> def make_pair(first, second): ...
>>> p = make_pair(1, 2)
```

```python
make_pair
    p
        pair_dict
            {'first': 1, 'second': 2}
```

**Dispatch Dictionaries**

```python
>>> p['first']
```

1

```python
make_pair
    p
        pair_dict
            {'first': 1, 'second': 2}
```

```python
p[‘first’]
```

1

```python
make_pair
    p
        pair_dict
            {'first': 1, 'second': 2}
```
DISPATCH DICTIONARIES

>>> p[‘first’] = 3

ANNOUNCEMENTS: MIDTERM 2

• Midterm 2 is on Wednesday, July 25.
  – Where? 2050 VLSB.
  – When? 7PM to 9PM.
• Closed book and closed electronic devices.
• One 8.5” x 11” ‘cheat sheet’ allowed.
• Group portion is 15 minutes long.
• If you have a conflict, please let us know by the end of today, July 23.

ANNOUNCEMENTS

• Homework 10 is due Tuesday, July 24.
• Project 3 is due Thursday, July 26.
• Homework 11 is due Friday, July 27.
Please ask for help if you need to. There is a lot of work in the weeks ahead, so if you are ever confused, consult (in order of preference) your study group and Piazza, your TAs, and Jom.
Don’t be clueless!

OOP IMPLEMENTATION: PREFACE

Things we will implement:
• Classes and objects
• Class variables
• Instance variables
• Methods
• Inheritance

OOP IMPLEMENTATION: PREFACE

Things we will not implement:
• Dot notation
  (We will see similar notation though.)
• Class methods
• Multiple inheritance
• Introspection
  (What class does this object belong to? What attributes does it have?)
OOP IMPLEMENTATION

We know that everything in Python is an object.

Classes and objects have attributes, which map names to values. These values can either be data or functions.

**Main idea:** We represent classes and objects as dictionaries that map attribute names to values.

---

OOP IMPLEMENTATION: CLASS VARIABLES

**First solution:**

```python
def make_class(attributes={}, base_class=None):
    return attributes

def make_pokemon_class():
    return make_class({'total_pokemon': 0})
```

```python
>>> Pokemon = make_pokemon_class()
>>> Pokemon['total_pokemon']
0
```

---

OOP IMPLEMENTATION: INHERITANCE

The current solution works. However, we need to modify it to include support for inheritance.

Why? If the current class does not have the class variable, its parent classes might have the variable.

We need to specify a way to recursively retrieve values for variables that may be defined in parent classes.

---

OOP IMPLEMENTATION: INHERITANCE

**Idea:** We create two inner functions, one that deals with locating the value for the class variable, and another that deals with setting new values for class attributes.

Our class now becomes a dispatch dictionary of at least two keys: get and set.
OOP IMPLEMENTATION: INHERITANCE

Modified solution:
def make_class(attributes={}, base_class=None):

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

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

A class is still a dictionary! The two new messages get and set allow us to use the general getter and setter functions.

To find the value of a class attribute...
...check if it’s already in the dictionary of attributes.

Or, if there is a parent class, check if the parent class has the class attribute.

We are passing messages to our classes.

OOP IMPLEMENTATION: OBJECTS

Just as we did with classes, we use dictionaries to represent objects:
def make_instance(cls):

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

An instance is a dictionary! The two messages get and set allow us to use the general getter and setter functions.

OOP IMPLEMENTATION: INHERITANCE

How do we use this new definition of a class?

```python
>>> Pokemon = make_pokemon_class()
>>> Pokemon['get']('total_pokemon')
... 

Find the general getter function in the dispatch dictionary... ...and use it to find the value of a certain class variable.
```

The new syntax is (unfortunately) more clunky. However, we can now access class variables from parent classes.

```python
>>> Pokemon = make_pokemon_class()
>>> def make_water_pokemon_class():
...     return make_class({}, Pokemon)
>>> WaterPokemon = make_water_pokemon_class()
>>> WaterPokemon['get']('total_pokemon')
... 

Find the general getter function in the dispatch dictionary... ...and use it to set or update the value of a certain class variable.
```
OOP IMPLEMENTATION: OBJECTS

How do we use this definition of an object?

```python
>>> Pokemon = make_pokemon_class()
>>> pikachu = make_instance(Pokemon)
>>> pikachu['set']['hp', 300]
```

Find the general setter function in the dispatch dictionary...

```python
>>> pikachu['get']['hp']
```

300

Find the general getter function in the dispatch dictionary...

REVIEW: BOUND METHODS

A method is bound to an instance.

```python
Pokemon.increase_hp(ashs_pikachu, 150)
```

is equivalent to

```python
ashs_pikachu.increase_hp(150)
```

BOUND METHODS

We have seen a variant of this idea before, during functional programming:

```python
def sum_of_squares(a, b):
    return a*a + b*b

def sum_with_25(b):
    return sum_of_squares(5, b)
```

BOUND METHODS

In the expression above, we do not pass in an object as the first argument, even though the definition of increase_hp seems to need it.

The first argument to the method increase_hp is already bound to the object ashs_pikachu.

OOP IMPLEMENTATION: LOOKING AHEAD

We would like to eventually have the following (revised) definition of the Pokemon class:

```python
def make_pokemon_class():
    def __init__(self, name, owner, hp):
        ...
    def increase_hp(self, amount):
        ...
```
OOP IMPLEMENTATION: OBJECTS

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)

    return get_value

OOP IMPLEMENTATION: OBJECTS

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

OOP IMPLEMENTATION: INSTANTIATION AND INITIALIZATION

We write a function init_instance that allows us to easily instantiate and initialize objects.

def init_instance(cls, *args):
    instance = make_instance(cls)
    init = cls['get']['__init__']
    if init is not None:
        init(instance, *args)
    return instance

OOP IMPLEMENTATION: INSTANTIATION AND INITIALIZATION

We have seen how we can implement instance variables, class variables, and methods.
We are almost done!

In Python's OOP system, the expression
Pokemon('Pikachu', 'Ash', 300)
both instantiated a new object and initialized its attributes with appropriate values.

The constructor method __init__ initialized the object.
OOP IMPLEMENTATION: USAGE

```python
def make_pokemon_class():
    def __init__(self, name, owner, hp):
        self['set']('name', name)
        self['set']('owner', owner)
        self['set']('hp', hp)
    def increase_hp(self, amount):
        old_hp = self['get']('hp')
        self['set']('hp', old_hp + amount)
    ...
    return make_class({'__init__': __init__,
                       'increase_hp': increase_hp, ...})

>>> Pokemon = make_pokemon_class()
>>> ashs_pikachu = Pokemon['new']('Pikachu', 'Ash', 300)
>>> ashs_pikachu['get']('hp')
300
>>> ashs_pikachu['get']('owner')
'Ash'
>>> ashs_pikachu['get']('increase_hp')(50)
>>> ashs_pikachu['get']('hp')
350
```

OOP IMPLEMENTATION: USAGE

CONCLUSION

Today, after five weeks of studying two major programming paradigms, we designed an OOP system from scratch using only dictionaries and functions!

**Main idea:** Classes and objects are “dispatch dictionaries”, which are passed messages, and which run code and assign (or update) variables as needed.