CS61A Lecture 20
Object-Oriented Programming: Implementation

Jom Magrotker
UC Berkeley EECS
July 23, 2012

Computer Science in the News

Researchers create memory with one bit per molecule

19 July 2012 | By Andrew Czyzewski

Print | Email | Share | Comments (2) | Save

In a major step forward for nanoscale computing, German and Japanese researchers have created magnetic memory with one bit per molecule.

That compares with current hard disks where one bit of digital information consists of around three million magnetic molecules.

The researchers' device consists of a single magnetic iron atom — which can be switched with an electric pulse — in the centre of an organic shell that protects the information stored within.

http://www.theengineer.co.uk/sectors/electronics/news/researchers-create-memory-with-one-bit-per-molecule/1013215.article

Computer Science in the News

Stanford Report, July 19, 2012

Stanford researchers produce first complete computer model of an organism

A mammoth effort has produced a complete computational model of the bacterium Mycoplasma genitalium, opening the door for biological computer-aided design.

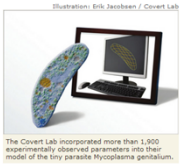
BY MAX MCCLURE

In a breakthrough effort for computational biology, the world's first complete computer model of an organism has been completed, Stanford researchers reported last week in the journal Cell.

A team led by Markus Covert, assistant professor of bioengineering, used data from more than 900 scientific papers to account for every molecular interaction that takes place in the life cycle of *Mycoplasma genitalium*, the world's smallest free-living bacterium.

By encompassing the entirety of an organism *in silico*, the paper fulfills a longstanding goal for the field. Not only does the model allow researchers to address questions that aren't practical to examine otherwise, it represents a stepping-stone toward the use of computer-aided design in bioengineering and medicine.

Illustration: Erik Jacobsen / Covert Lab



The Covert Lab incorporated more than 1,900 experimentally observed parameters into their model of the tiny parasite Mycoplasma genitalium.

http://news.stanford.edu/news/2012/july/computer-model-organism-071812.html

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.

Review: Dispatch Functions

```
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*.

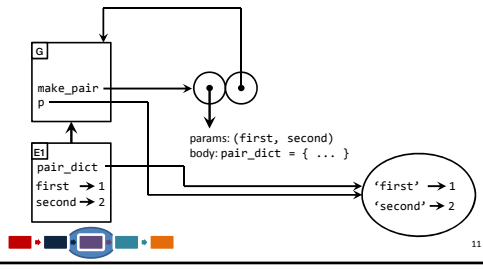
Dispatch Dictionaries

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

How do we create and use "pair objects" now?

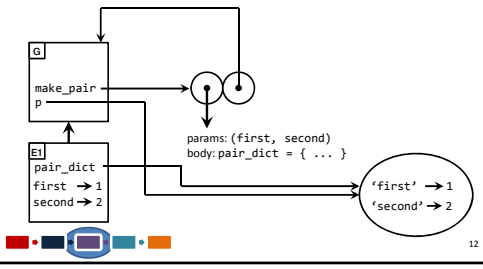
Dispatch Dictionaries

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



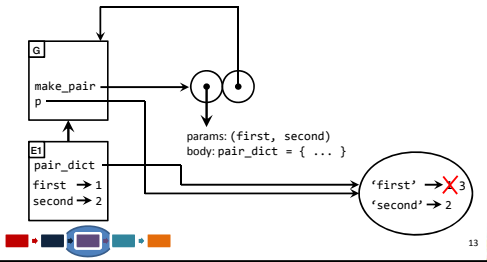
Dispatch Dictionaries

```
>>> p['first']
1
```



Dispatch Dictionaries

```
>>> p['first'] = 3
```



Announcements: Midterm 2

- Midterm 2 is on **Wednesday, July 25**.
  - *Where*? 2050 VLSB.
  - *When*? 7PM to 9PM.
  - *How much*? Material covered until, and including, July 19.
- 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

We have discovered how we can create a "pair object" using just functions and dictionaries.

We will now use many of the same ideas to explore how an object-oriented programming system can be constructed from scratch.

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

We start simple and first implement *class variables*. For example, we want to mimic

```
class Pokemon:
    total_pokemon = 0
```

`total_pokemon` is a *class variable*.
We access its value using the expression `Pokemon.total_pokemon`.

OOP Implementation: Class Variables

First solution:

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

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

Dictionary of class variables

OOP Implementation: Class Variables

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

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

>>> 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
```

To find the value of a class attribute…

… check if it is already in the dictionary of attributes.

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

A class is still a dictionary! The two new 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?

```
>>> Pokemon = make_pokemon_class()
>>> Pokemon['get']('total_pokemon')
0
>>> Pokemon['set']('total_pokemon', 1)
```

Find the general getter function in the dispatch dictionary…

… and use it to *find* the value of a certain class variable.

Find the general setter function in the dispatch dictionary…

… and use it to *set* or *update* the value of a certain class variable.

OOP Implementation: Inheritance

`Pokemon['get']('total_pokemon')`
is equivalent to
`Pokemon.total_pokemon`

`Pokemon['set']('total_pokemon', 1)`
is equivalent to
`Pokemon.total_pokemon = 1`

We are passing messages to our classes.

OOP Implementation: Inheritance

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

```
>>> Pokemon = make_pokemon_class()
>>> def make_water_pokemon_class():
        return make_class({}, Pokemon)
>>> WaterPokemon = make_water_pokemon_class()
>>> WaterPokemon['get']('total_pokemon')
0
```

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

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

What class is this object an instance of?

To find the value of an instance attribute…

… check if it is already in the dictionary of attributes.

Otherwise, check if the class has a value for the attribute.

Dictionary of instance attributes

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

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

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

Every instance gets its own dictionary of instance attributes.

OOP Implementation: Objects

How do we use this definition of an object?

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

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

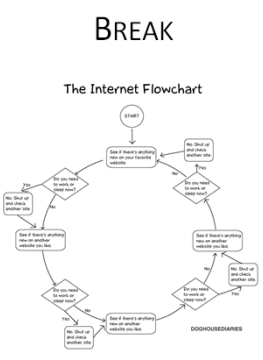
... and use it to *set* or *update* the value of a certain instance variable.

```
>>> pikachu['get']('hp')
300
```

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

... and use it to *find* the value of a certain instance variable.

Break



Review: Bound Methods

A method is *bound* to an instance.

Use the `increase_hp` method defined for objects *of* the Pokemon class...

... with `self` being the `ashs_pikachu` object ...

... and amount being `150`.

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

"of"

is equivalent to

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

Use the `increase_hp` method *of* (or *bound* to) the `ashs_pikachu` object...

... with `amount` being `150`.

Bound Methods

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

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`.

Bound Methods

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

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

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

Notice that we do not need to provide the first argument anymore! a is *bound* to 5.

OOP Implementation: Looking Ahead

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

```
def make_pokemon_class():
    def __init__(self, name, owner, hp):
        ...
    def increase_hp(self, amount):
        ...
```

These `self` arguments need to be bound to the object on which the method is called.

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)
```

All the methods are defined in the class. We might get one of these: if we do, bind the first argument to the current instance to produce a bound method for the instance.

OOP Implementation: Objects

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

`callable` is a predicate that checks if the argument provided can be called with arguments.

If the value provided is callable (for example, a function), ...

... make the new bound method...

... where the first argument is bound to the instance given, and all the other arguments remain the same...

... and return this bound method.

If the value provided is not callable, return it as is.

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: 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
```

Make an instance of the class.

Find the constructor method.

If the constructor exists...

... use it to initialize the object.

Return the new instance.

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
```

The constructor name is fixed here.

OOP Implementation: Instantiation and Initialization

We add an extra message that our classes can understand, which will allow us to create new objects:

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

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

OOP Implementation: Usage

```
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, ...})
```

OOP Implementation: Usage

```
>>> 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
```

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.