CS61A Lecture 14

Object-Oriented Programming

Jom Magrotker
UC Berkeley EECS
July 11, 2012
Is that smile real or fake?
A computerized system developed at MIT can tell the difference between smiles of joy and smiles of frustration.
David L. Chandler, MIT News Office

May 25, 2012

Is that smile real or fake?
A computerized system developed at MIT can tell the difference between smiles of joy and smiles of frustration.

TODAY

• Review: Object-Oriented Programming
• Inheritance
WHERE WE WERE: FUNCTIONAL PROGRAMMING

One of many programming paradigms where functions are the central players.

Functions can work on data, which can sometimes be other functions.
WHERE WE WERE: FUNCTIONAL PROGRAMMING

Key Feature: Composition

Functions can receive input from, and provide output to, other functions.

Many problems can be solved by “chaining” the outputs of one function to the inputs of the next.
WHERE WE WERE: FUNCTIONAL PROGRAMMING

Key Feature: Composition

Example: Sum of all prime numbers from 2 to 100.

1. Generate all the numbers from 2 to 100.
2. Filter out the prime numbers.
3. Sum the prime numbers up.

\[
\text{reduce}(\text{add}, \text{filter}(\lambda\text{num}: \text{is\_prime}(\text{num}), \text{range}(2, 101)), 0)
\]
WHERE WE WERE: FUNCTIONAL PROGRAMMING

Key Feature: Statelessness

Functions always produce the same outputs for the same inputs.

This makes them incredibly useful for, for example, processing a lot of data in parallel.
WHERE WE WERE: FUNCTIONAL PROGRAMMING

Key Feature: Statelessness

Because of statelessness, if we need to update a piece of data, we need to create a whole new piece of data.

my_idict = idict_insert(my_idict, 'a', 5)
WHERE WE WERE: FUNCTIONAL PROGRAMMING

Key Feature: Statelessness

Functional programming is clunky for data that changes over time, or that has state.

We need an easier way to work with stateful data.
OBJECT-ORIENTED PROGRAMMING

A new programming paradigm where **objects** are the central players.

*Objects* are data structures that are combined with associated behaviors. They are “smart bags” of data that have state and can interact.

Functions can do one thing; objects can do many related things.
**TERMINOLOGY**

Any person is a **human**.

A **single person** has a **name** and **age**.

- **OBJECT**
- **CLASS**

**INSTANCE of the Human class**

**INSTANCE VARIABLES**
**TERMINOLOGY**

An **object** is an **instance** of a **class**. For example, a person is an instance of a human.

The class describes its objects: it is a **template**.
Objects and instance variables have a “has-a” relationship.

An instance variable is an attribute specific to an instance.

An object has an instance variable.
TERMINOLOGY

A single person can eat and sleep.

The population of the Earth is 7 billion.
Objects have certain behaviors, known as *methods*.

There are attributes for the class as a whole, not for specific instances: these are *class variables*. 
ANNOUNCEMENTS: MIDTERM 1

• Grades are available through glookup.
• Mean: 38.0, standard deviation: 8.6.
• Your TA will distribute the graded midterms in lab today, in exchange for a completed survey.
• The average of the class improved by 1 point when scores on the group portion were considered.
• Midterm solutions will be released soon.
• Post-midterm destress potluck tonight from 6:30pm to 10pm in the Wozniak lounge (4th floor Soda).
OOP IN PYTHON

class Pokemon:

    __total_pokemon = 0  # CLASS VARIABLE
    def __init__(self, name, owner, hit_pts):
        self.__name = name
        self.__owner = owner
        self.__hp = hit_pts
        Pokemon.__total_pokemon += 1
class Pokemon:
    __total_pokemon = 0
    def __init__(self, name, owner, hit_pts):
        self.__name = name
        self.__owner = owner
        self.__hp = hit_pts
        Pokemon.__total_pokemon += 1

CONSTRUCTOR

self refers to the INSTANCE.

Class variables are referenced using the name of the class, since they do not belong to a specific instance.
class Pokemon:
    __total_pokemon = 0
    def __init__(self, name, owner, hit_pts):
        self.__name = name
        self.__owner = owner
        self.__hp = hit_pts
        Pokemon.__total_pokemon += 1
OOP IN PYTHON

class Pokemon:

    ...  

    def increase_hp(self, amount):
        self.__hp += amount

    def decrease_hp(self, amount):
        self.__hp -= amount
OOP in Python

class Pokemon:
    ...
    def increase_hp(self, amount):
        self.__hp += amount
    def decrease_hp(self, amount):
        self.__hp -= amount

Every method needs self as an argument.
class Pokemon:
    ...
def get_name(self):
    return self.__name
def get_owner(self):
    return self.__owner
def get_hit_pts(self):
    return self.__hp
OOP in Python

```python
>>> ashes_pikachu = Pokemon('Pikachu', 'Ash', 300)
>>> mistys_togepi = Pokemon('Togepi', 'Misty', 245)
>>> mistys_togepi.get_owner()
'Misty'
>>> ashes_pikachu.get_hit_pts()
300
>>> ashes_pikachu.increase_hp(150)
>>> ashes_pikachu.get_hit_pts()
450
```

We now have state! The same expression evaluates to different values.
The statement
ashs_pikachu = Pokemon(‘Pikachu’, ‘Ash’, 300)

*instantiates* a new object.

The *__init__* method (the *constructor*) is called by this statement.

Objects can *only* be created by the constructor.
SMART BAGS OF DATA

ashs_pikachu = Pokemon(‘Pikachu’, ‘Ash’, 300)
mistys_togepi = Pokemon(‘Togepi’, ‘Misty’, 245)

The statements above create two new objects:

- **Instance variables:**
  - __name
  - __owner
  - __hp

- **Methods:**
  - increase_hp
  - decrease_hp
  - get_name
  - get_owner
  - get_hit_pts
SMART BAGS OF DATA

Each object gets its own set of instance variables and **bound** methods.

Each object is a “smart bag” of data: it has data and it can also manipulate the data.

---

**Instance variables:**
__name__
__owner__
__hp__

**Methods:**
increase_hp
decrease_hp
get_name
get_owner
get_hit_pts
A method is *bound* to an instance.

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

*Self is implicitly the object itself.*
Object Identity

Every object has its own set of independent instance variables and bound methods.

```python
>>> ash_pikachu = Pokemon('Pikachu', 'Ash', 300)
>>> brocks_pikachu = ash_pikachu
>>> brocks_pikachu is ash_pikachu
True
``` 

Assigning the same object to two different variables.

```python
>>> brocks_pikachu = Pokemon('Pikachu', 'Brock', 300)
>>> brocks_pikachu is ash_pikachu
False
``` 

The `is` operator checks if the two variables evaluate to the same object.
OOP in Python: Practice

Which methods in the Pokemon class should be modified to ensure that the HP never goes down below zero? How should it be modified?

We modify the decrease_hp method:

```python
def decrease_hp(self, amount):
    self.__hp -= amount
    if self.__hp < 0:
        self.__hp = 0
```
OOP IN PYTHON: PRACTICE

Write the method attack that takes another Pokemon object as an argument. When this method is called on a Pokemon object, the object screams (= prints) its name and reduces the HP of the opposing Pokemon by 50.

```python
>>> mistys_togepi.get_hp()
245
>>> ashs_pikachu.attack(mistys_togepi)
Pikachu!
>>> mistys_togepi.get_hp()
195
```
OOP IN PYTHON: PRACTICE

Write the method `attack` that takes another Pokemon object as an argument. When this method is called on a Pokemon object, the object screams (= prints) its name and reduces the HP of the opposing Pokemon by 50.

```python
def attack(self, other):
    print(self.get_name() + "!")
    other.decrease_hp(50)
```
A NOTE ABOUT DOUBLE UNDERSCORES

All instance variables so far were preceded by double underscores. *This is not necessary!*

The double underscores tell Python, and other Python programmers, that this variable is not to be used *directly* outside the class.

It is necessary for `__init__` though (which is not an instance variable)! We will see why tomorrow.
A NOTE ABOUT DOUBLE UNDERSCORES

Python will *modify* the name of the variable so that you cannot use it directly outside the class.

You *can* find out the new name (using `dir`), but if you need this extra effort, you are either debugging your code or doing something wrong.
A NOTE ABOUT DOUBLE UNDERSCORES

The code from before could have been

class Pokemon:
    total_pokemon = 0
    def __init__(self, name, owner, hit_pts):
        self.name = name
        self.owner = owner
        self.hp = hit_pts
        Pokemon.total_pokemon += 1
A NOTE ABOUT DOUBLE UNDERSCORES

We can then obtain the attributes more directly.

```python
>>> ash_pikachu = Pokemon('Pikachu', 'Ash', 300)
>>> mistys_togepi = Pokemon('Togepi', 'Misty', 245)
>>> mistys_togepi.owner
'Misty'
>>> ash_pikachu.hp
300
>>> ash_pikachu.increase_hp(150)
>>> ash_pikachu.hp
450
```
PROPERTIES

Python allows us to create attributes that are computed from other attributes, but need not necessarily be instance variables.

Say we want each Pokemon object to say its complete name, constructed from its owner’s name and its own name.
PROPERTIES

One way is to define a new *method*.

class Pokemon:
    ...
    def complete_name(self):
        return self.owner + "'s " + self.name

>>> ash_pikachu.complete_name()
‘Ash’s Pikachu’

However, this seems like it should be an attribute (something the data *is*), instead of a method (something the data can *do*).
Another way is to use the property *decorator*.

class Pokemon:
    ...
    @property
    def complete_name(self):
        return self.owner + "'s " + \ 
        self.name

>>> ashs_pikachu.complete_name
‘Ash’s Pikachu’
BREAK

CHOP CHOP

CHOP AARGH

WHAT IS ALL THIS CRAP?

WHY IS THIS STRUCTURE HERE?

WHERE COULD THIS BRIDGE POSSIBLY LEAD?

THIS SIGN DOESN'T HELP ME MUCH.

WHAT A HORRIBLY DESIGNED STREET, MOST INEFFICIENT.

GOOD GOD! WHAT THE HELL DOES THIS CONTRAPTION DO?

I hate reading other people's code.

http://abstrusegoose.com/432
Occasionally, we find that many abstract data types are related.

For example, there are many different kinds of people, but all of them have similar methods of eating and sleeping.
INHERITANCE

We would like to have different kinds of Pokémon, which differ (among other things) in the amount of points lost by its opponent during an attack.

The only method that changes is attack. All the other methods *remain the same*. Can we avoid *duplicating code* for each of the different kinds?
INHERITANCE

Key OOP Idea: Classes can inherit methods and instance variables from other classes

class WaterPokemon(Pokemon):
    def attack(self, other):
        other.decrease_hp(75)

class ElectricPokemon(Pokemon):
    def attack(self, other):
        other.decrease_hp(60)
INHERITANCE

Key OOP Idea: Classes can inherit methods and instance variables from other classes

class WaterPokemon(Pokemon):
    def attack(self, other):
        other.decrease_hp(75)

class ElectricPokemon(Pokemon):
    def attack(self, other):
        other.decrease_hp(60)
INHERITANCE

Key OOP Idea: Classes can inherit methods and instance variables from other classes

class WaterPokemon(Pokemon):
    def attack(self, other):
        other.decrease_hp(75)

class ElectricPokemon(Pokemon):
    def attack(self, other):
        other.decrease_hp(60)

The attack method from the Pokemon class is *overridden* by the attack method from the WaterPokemon class.
INHERITANCE

```python
>>> ash_squirtle = WaterPokemon('Squirtle', 'Ash', 314)
>>> misty_togepi = Pokemon('Togepi', 'Misty', 245)
>>> misty_togepi.attack(ash_squirtle)
>>> ash_squirtle.get_hit_pts()
264
>>> ash_squirtle.attack(misty_togepi)
>>> misty_togepi.get_hit_pts()
170
```
INHERITANCE

>>> ashs_squirtle = WaterPokemon('Squirtle', 'Ash', 314)
>>> mistys_togepi = Pokemon('Togepi', 'Misty', 245)
>>> mistys_togepi.attack(ashs_squirtle)
>>> ashs_squirtle.get_hit_pts()
264
>>> ashs_squirtle.attack(mistys_togepi)
>>> mistys_togepi.get_hit_pts()
170

mistys_togepi uses the attack method from the Pokemon class.
INHERITANCE

```python
>>> ashs_squirtle = WaterPokemon('Squirtle', 'Ash', 314)
>>> mistys_togepi = Pokemon('Togepi', 'Misty', 245)
>>> mistys_togepi.attack(ashs_squirtle)
>>> ashs_squirtle.get_hit_pts()
264
>>> ashs_squirtle.attack(mistys_togepi)
>>> mistys_togepi.get_hit_pts()
170
```

ashs_squirtle uses the attack method from the WaterPokemon class.
INHERITANCE

```python
>>> ashs_squirtle = WaterPokemon('Squirtle', 'Ash', 314)
>>> mistys_togepi = Pokemon('Togepi', 'Misty', 245)
>>> mistys_togepi.attack(ashs_squirtle)
>>> ashs_squirtle.get_hit_pts()
264
>>> ashs_squirtle.attack(mistys_togepi)
>>> mistys_togepi.get_hit_pts()
170
```
INHERITANCE: WHAT HAPPENS HERE?

class ElectricPokemon(Pokemon):
    def __init__(self, name, owner, hp, origin):
        self.__origin = origin

ashs_pikachu = ElectricPokemon('Pikachu', 'Ash', 300, 'Pallet Town')
ashs_pikachu.get_hit_pts()
Inheritance: What Happens Here?

One fix is to first call the constructor of the superclass. The constructor of the subclass overrode the constructor of the superclass, which is why the other instance variables were never assigned.

class ElectricPokemon(Pokemon):
    def __init__(self, name, owner, hp, origin):
        Pokemon.__init__(self, name, owner, hp)
        self.__origin = origin
CONCLUSION

• Object-oriented programming is another paradigm that makes objects its central players, not functions.
• Objects are pieces of data and the associated behavior.
• Classes define an object, and can inherit methods and instance variables from each other.
• *Preview*: If it looks like a duck and quacks like a duck, is it a duck?