CS61A Lecture 14
Object-Oriented Programming

Jom Magrotker
UC Berkeley EECS

July 11, 2012

Computer Science in the News

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

http://web.mit.edu/newsoffice/2012/smile-detector-0525.html

2

Today

- Review: Object-Oriented Programming
- Inheritance

3

Where We Were: Functional Programming

Style of programming

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

Functions can work on data, which can sometimes be other functions.

4

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.

5

Where We Were: Functional Programming

Key Feature: Composition

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

3. Sum the prime numbers up.

2. Filter out the prime numbers.

1. Generate all the numbers from 2 to 100.

```
reduce(add,
       filter(lambda num: is_prime(num),
              range(2, 101)),
       0)
```

6

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*.

7

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.

2. Insert the new item.

```
my_idict = idict_insert(my_idict, 'a', 5)
```

4. Store the new IDict. 3. Get a new IDict. 1. Take an IDict.

8

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.

9

Object-Oriented Programming

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

Hence the term "object-oriented".

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.

10

Terminology

Object Class

Any person is a human.

A single person has a name and age.

Instance of the Human class Instance Variables

11

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*.

12

Terminology

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

Methods

A single person can **eat** and **sleep**.

The **population** of the Earth is 7 billion.

Class Variables

Terminology

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

```
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
```

Class Variable

Instance Variables

http://pkmnhackersonline.com/wp-content/uploads/2012/06/pokemon-logo.jpg

OOP in Python

```
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
```

`self` refers to the Instance.

Constructor

Class variables are referenced using the *name of the class*, since they do not belong to a specific instance.

OOP IN PYTHON

```
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
```

"of"

Name *of* the instance (self)

Total number *of* Pokémon

OOP IN PYTHON

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

METHODS

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.

OOP IN PYTHON

```
class Pokemon:
    ...
    def get_name(self):
        return self.__name
    def get_owner(self):
        return self.__owner
    def get_hit_pts(self):
        return self.__hp
```

SELECTORS

OOP IN PYTHON

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

We now have state!
The same expression
evaluates to different values.

http://media.photobucket.com/image/pikachu%20surprised/Romantic_Princess/l_7b0cb7ca3a96b9e3350eef6a2c99f981.gif?o=2

OOP IN PYTHON

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

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

Instance variables:
__name
__owner
__hp

Methods:
increase_hp
decrease_hp
get_name
get_owner
get_hit_pts

Bound Methods

A method is *bound* to an instance.

`Pokemon.increase_hp(ashs_pikachu, 150)`

evaluates to

`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.

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

Assigning the same object to two different variables.

The `is` operator checks if the two variables evaluate to the same object.

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

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:

```
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 Pokémon by 50.

```
>>> 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 Pokémon by 50.

```
def attack(self, other):
    print(self.get_name() + “!”)
    other.decrease_hp(50)
```

31

A Note About Double Underscores

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

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

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

32

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.

http://images.wikia.com/poohadventures/images/5/5d/Brock.gif

33

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
```

34

A Note About Double Underscores

We can then obtain the attributes more directly.

```
>>> ashs_pikachu = Pokemon(‘Pikachu’, ‘Ash’, 300)
>>> mistys_togepi = Pokemon(‘Togepi’, ‘Misty’, 245)
>>> mistys_togepi.owner
‘Misty’
>>> ashs_pikachu.hp
300
>>> ashs_pikachu.increase_hp(150)
>>> ashs_pikachu.hp
450
```

35

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.

36

Properties

One way is to define a new *method*.

```
class Pokemon:
    ...
    def complete_name(self):
        return self.owner + “’s ” + \
               self.name

>>> ashs_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*).

Properties

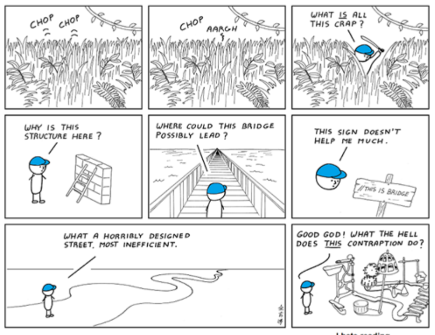
Another way is to use the `property` *decorator*.

```
class Pokemon:
    ...
    @property
    def complete_name(self):
        return self.owner + “’s ” + \
               self.name

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

A new attribute is calculated!

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.
//THIS IS BRIDGE
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

Inheritance

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.

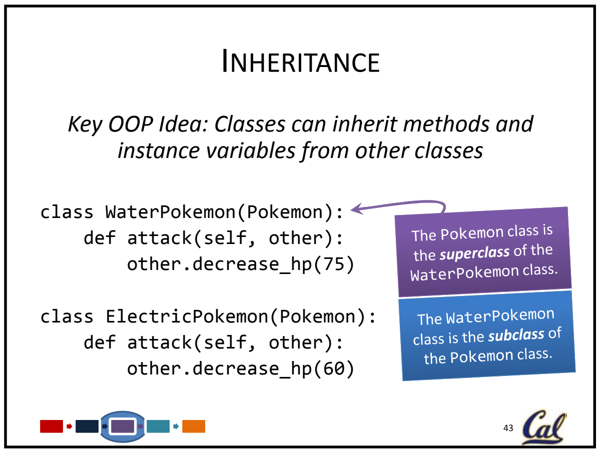
The only method that changes is `attack`. All the other methods *remain the same*. Can we avoid *duplicating code* for each of the different kinds?

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

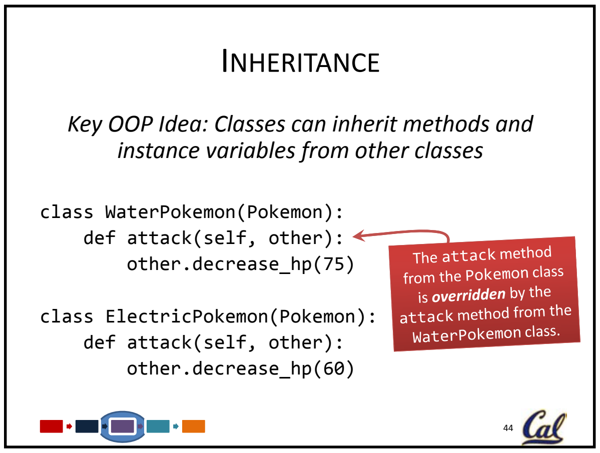
```
class WaterPokemon(Pokemon):
    def attack(self, other):
        other.decrease_hp(75)

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

The Pokemon class is the ***superclass*** of the WaterPokemon class.

The WaterPokemon class is the ***subclass*** of the Pokemon class.

43



INHERITANCE

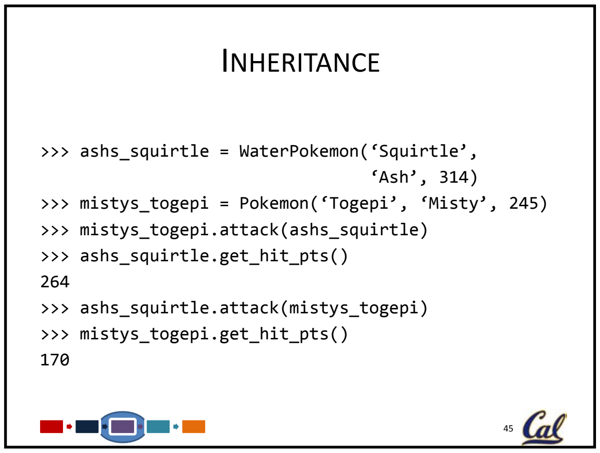
Key OOP Idea: Classes can inherit methods and instance variables from other classes

```
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.

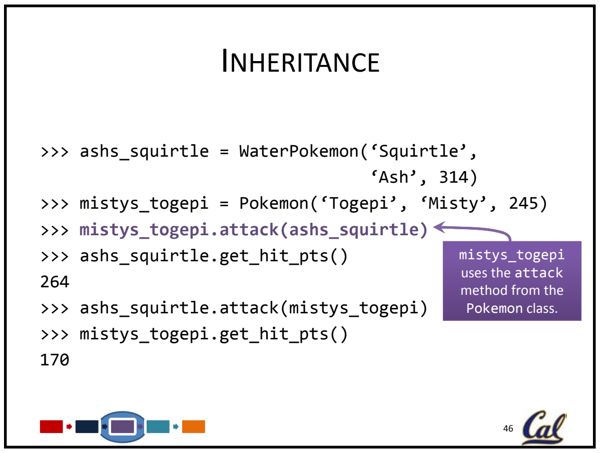
44



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
```

45

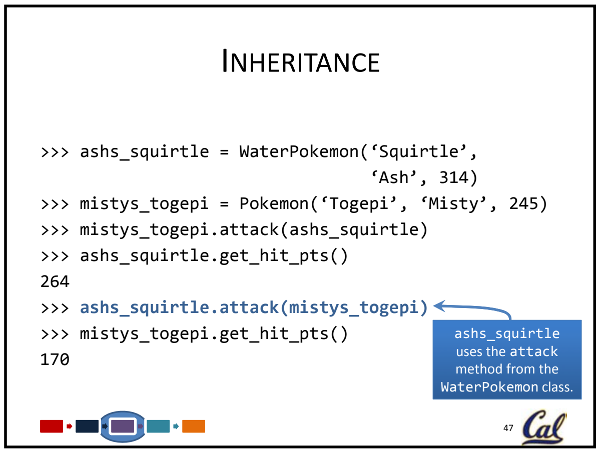


INHERITANCE

```
>>> 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.

46

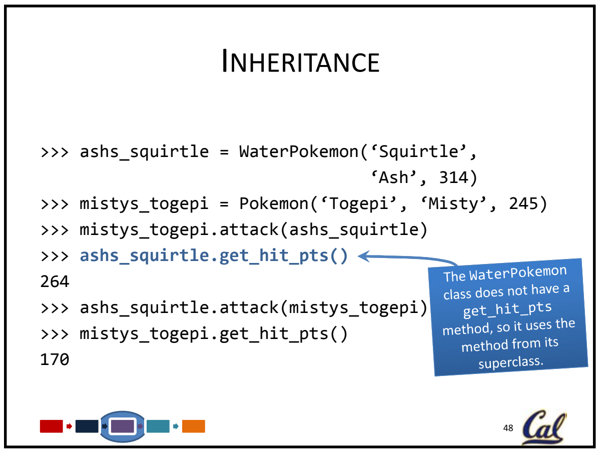


INHERITANCE

```
>>> 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
```

ashs_squirtle uses the attack method from the WaterPokemon class.

47



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
```

The WaterPokemon class does not have a get_hit_pts method, so it uses the method from its superclass.

48

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?