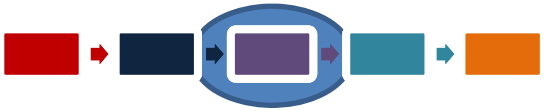
CS61A Lecture 15
Object-Oriented Programming, Mutable Data Structures

Jom Magrotker
UC Berkeley EECS

July 12, 2012



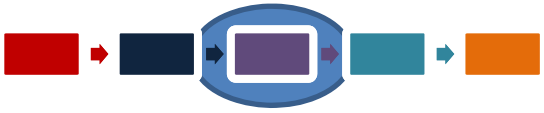
Computer Science in the News

Left: An illustration of the EyeMusic SSD, showing a user with a camera mounted on the glasses, and scalp headphones, hearing musical notes that create a mental image of the visual scene in front of him. He is reaching for the red apple in a pile of green ones. Top right: close-up of the glasses-mounted camera and headphones; bottom right: hand-held camera pointed at the object of interest. [Image credit Maxim Dupliy, Amir Amedi and Shelly Levy-Tzedek]

http://www.iospress.nl/ios_news/music-to-my-eyes-device-converting-images-into-music-helps-individuals-without-vision-reach-for-objects-in-space/

TODAY

- Review: Inheritance
- Polymorphism
- Mutable Lists



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)
```

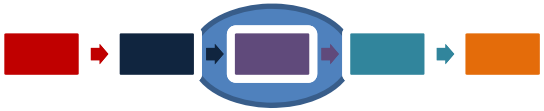
Use the `increase_hp` method *of* (or *bound* to) the `ashs_pikachu` object…

… with `amount` being `150`.

Review: 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.



Review: Inheritance

We would like to have different kinds of Pokémon, besides the "Normal" Pokemon (like Togepi) 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?

Review: 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)
```

Review: Inheritance

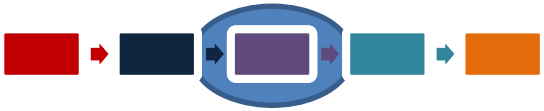
Key OOP Idea: Classes can inherit methods and instance variables from other classes.

```
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*** (or ***parent class***) of the WaterPokemon class.

The WaterPokemon class is the ***subclass*** (or ***child class***) of the Pokemon class.



Review: 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.

REVIEW: 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
```

REVIEW: 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.

REVIEW: 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.

Review: 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.

Review: Inheritance

If the class of an object has the method or attribute of interest, that particular method or attribute is used.

Otherwise, the method or attribute of its parent is used.

Inheritance can be many levels deep.

If the parent class does not have the method or attribute, we check the parent of the parent class, and so on.

Review: Inheritance

We can also both *override* a parent's method or attribute *and* use the original parent's method.

Say that we want to modify the attacks of Electric Pokémon: when they attack another Pokémon, the other Pokémon loses the *original* 50 HP, but the Electric Pokémon gets an increase of 10 HP.

Review: Inheritance

```
class ElectricPokemon(Pokemon):
    ...
    def attack(self, other):
        Pokemon.attack(self, other)
        self.increase_hp(10)
```

We use the original attack method from the parent.

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.

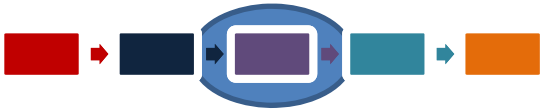
PROPERTIES

One way is to define a new *method*.

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

>>> 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!

ANNOUNCEMENTS

- Project 2 is due **Friday, July 13**.
- Homework 7 is due **Saturday, July 14**.
- Project 3 will be released this weekend, due **Tuesday, July 24**.
- Your TA will distribute your graded midterms in exchange for a completed survey.

What is Your Type?

OOP enables us to easily make new abstract data types for different types of data.

```
>>> type(ashs_pikachu)
<class 'ElectricPokemon'>
>>> type(ashs_squirtle)
<class 'WaterPokemon'>
>>> type(mistys_togepi) is Pokemon
True
```

The type function displays the type of its argument.

POLYMORPHISM

Write the method `attack_all` for the Pokemon class that takes a tuple of Pokemon objects as its argument. When called on a Pokemon object, that Pokémon will attack each of the Pokémon in the provided tuple. (Ignore the output printed by the `attack` method.)

```
>>> ashs_pikachu = ElectricPokemon(‘Pikachu’, ‘Ash’, 300)
>>> ashs_squirtle = WaterPokemon(‘Squirtle’, ‘Ash’, 314)
>>> mistys_togepi = Pokemon(‘Togepi’, ‘Misty’, 245)
>>> mistys_togepi.attack_all((ashs_pikachu, ashs_squirtle))
>>> ashs_pikachu.get_hit_pts()
250
>>> ashs_squirtle.get_hit_pts()
264
```

POLYMORPHISM

```
class Pokemon:
    ...
    def attack_all(self, others):
        for other in others:
            self.attack(other)
```

POLYMORPHISM

```
for other in others:
    self.attack(other)
```

other can be an object of many different data types: `ElectricPokemon`, `WaterPokemon`, and `Pokemon`, for example.

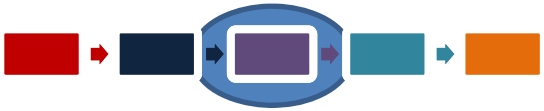
POLYMORPHISM

```
for other in others:
    self.attack(other)
```

`attack` can work on objects of many *different* data types, without having to consider each data type separately.

`attack` is *polymorphic*.

poly = Many
morph = Forms



POLYMORPHISM

Write the method `attacked_by` for the Pokemon class that takes a tuple of Pokemon objects as its argument. When called on a Pokemon object, that Pokémon will be attacked by each of the Pokémon in the provided tuple.

```
>>> ashs_pikachu = ElectricPokemon(‘Pikachu’, ‘Ash’, 300)
>>> ashs_squirtle = WaterPokemon(‘Squirtle’, ‘Ash’, 314)
>>> mistys_togepi = Pokemon(‘Togepi’, ‘Misty’, 245)
>>> ashs_squirtle.attacked_by((ashs_pikachu, mistys_togepi))
>>> ashs_squirtle.get_hit_pts()
204
```

POLYMORPHISM

```
class Pokemon:
    ...
    def attacked_by(self, others):
        for other in others:
            other.attack(self)
```

POLYMORPHISM

```
for other in others:
    other.attack(self)
```

other can be an object of many different data types: `ElectricPokemon`, `WaterPokemon`, and `Pokemon`, for example.

It can also be an object of *any other class* that has an `attack` method.

POLYMORPHISM

Key OOP idea: The same method can work on data of *different types*.

We have seen a polymorphic function before:

```
>>> 3 + 4
7
>>> ‘hello’ + ‘ world’
‘hello world’
```

The + operator, or the add function, is *polymorphic*. It can work with both numbers and strings.

Polymorphism: How Does + Work?

We try the `type` function on other expressions:

```
>>> type(9001)
<class ‘int’>
>>> type(‘hello world’)
<class ‘str’>
>>> type(3.0)
<class ‘float’>
```

Wait!

Are these *objects* of different classes?

http://diaryofamadgayman.files.wordpress.com/2012/01/wait-what.jpg

Polymorphism: How Does + Work?

Everything (even numbers and strings) in Python is an object.

Not true for many other languages.

In particular, every class (`int`, `str`, `Pokemon`, ...) is a subclass of a built-in `object` class.

This is *not* the term 'object', but a class whose *name* is `object`.

Everything is an Object

The `dir` function shows the attributes of a class:

```
>>> dir(WaterPokemon)
['__class__', '__delattr__', '__dict__', ...,
'attack', 'decrease_hp', 'get_hit_pts', 'get_name',
'get_owner', 'increase_hp', 'total_pokemon']
```

The green attributes were defined inside (and inherited from) the Pokemon class and the blue attribute was defined directly inside the `WaterPokemon` class.

Where did the red attributes come from?

Everything is an Object

The `object` class provides extra attributes that enable polymorphic operators like + to work on many different forms of data.

```
>>> dir(object)
['__class__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce
'__repr__', '__setattr__', '__size
'__str__', '__subclasshook__']
```

We will consider a few of these. The rest, we leave to your experiments.

Everything is an Object

Since *everything is an object*, this means we should be able to call methods on everything, including numbers:

```
>>> 3 + 4
7
>>> (3).__add__(4)
7
```

Everything is an Object

Python converts the expression

```
3 + 4
```

to

```
(3).__add__(4)
```

The conversion is slightly more complicated than this, but the basic idea is the same.

3 is an `int` object.

The `int` class has an `__add__` method.

Everything is an Object

Python converts the expression

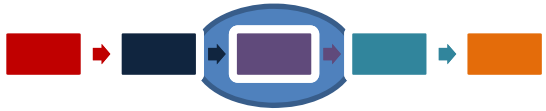
`‘hello’ + ‘ world’`

to

`(‘hello’).__add__(‘ world’)`

‘hello’ is a str object.

The str class *also* has a *different* __add__ method.



Everything is an Object

If a class has an `__add__` method, we can use the + operator on two objects of the class.

Similarly, for example,

`4 > 3` is converted to `(4).__gt__(3)`

`4 <= 3` is converted to `(4).__le__(3)`

`4 == 3` is converted to `(4).__eq__(3)`

`(4).__str__()` produces the string representation of 4 (used in printing, for example).

EVERYTHING IS AN OBJECT

Implement the method `__gt__` for the Pokemon class that will allow us to check if one Pokémon is "greater" than another, which means that it has (strictly) more HP than the other.

```
>>> ashs_pikachu = ElectricPokemon('Pikachu',
                                   'Ash', 300)
>>> ashs_squirtle = WaterPokemon('Squirtle',
                                 'Ash', 314)
>>> ashs_pikachu > ashs_squirtle
False
```

Everything is an Object

Implement the method `__gt__` for the Pokemon class that will allow us to check if one Pokémon is "greater" than another, which means that it has (strictly) more HP than the other.

```
class Pokemon:
    ...
    def __gt__(self, other):
        return ____________________________________________
```

Everything is an Object

Implement the method `__gt__` for the Pokemon class that will allow us to check if one Pokémon is "greater" than another, which means that it has (strictly) more HP than the other.

```
class Pokemon:
    ...
    def __gt__(self, other):
        return self.get_hit_pts() > other.get_hit_pts()
```

Everything is an Object

The `object` class provides a default `__gt__` method, but the `__gt__` method we defined for the `Pokemon` class *overrides* the method provided by the `object` class, as we would expect from inheritance.

Bottom line:

Inheritance and polymorphism in Python OOP allow us to *override standard operators*!

Everything is an Object

Side-note:

Every class inherits from the `object` class, including the `Pokemon` class.

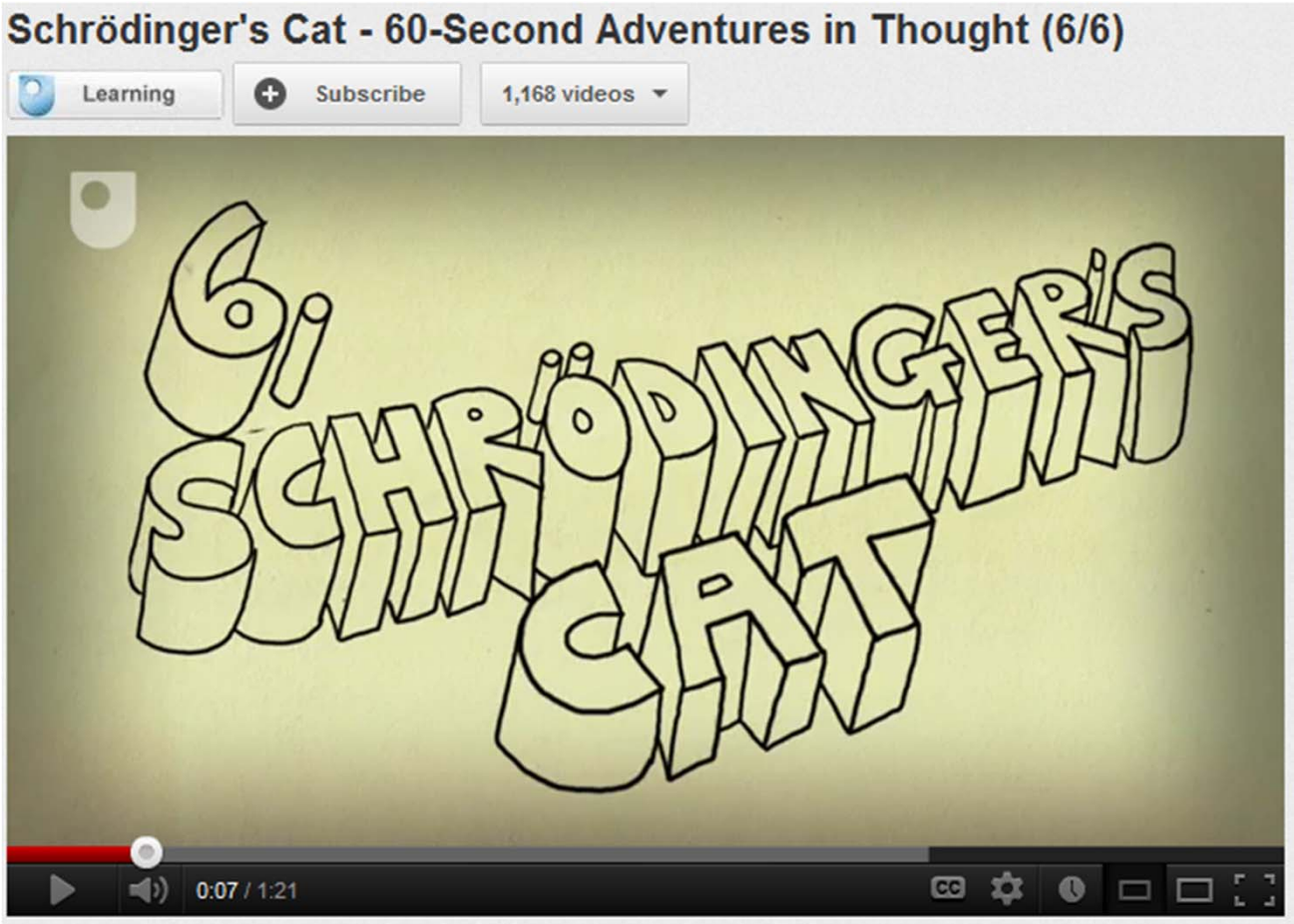
```
class Pokemon:
```

is shorthand for

```
class Pokemon(object):
```

Both are equivalent and correct.

Video Break



http://www.youtube.com/watch?v=d1tn56vWU_g

Mutable Data Structures

We have seen that objects possess state: they can change over time. Objects are thus ***mutable***.

The IRLists and IDicts we saw earlier were *im*mutable, which means that once created, they could not be modified.

Python has built-in list and dictionary data structures that *are* mutable.

Python Lists: A Primer

```
>>> a = [3, 4, 5]
>>> a[0]
3
>>> a[1:]
[4, 5]
>>> a[0] = 7
>>> a
[7, 4, 5]
```

Slicing makes a new list.

Could not have done this with tuples!

Python Lists: A Primer

```
>>> a
[7, 4, 5]
>>> a.append(10)
>>> a
[7, 4, 5, 10]
>>> a.extend([2, 3])
>>> a
[7, 4, 5, 10, 2, 3]
>>> a[2:4] = [6, 8]
[7, 4, 6, 8, 2, 3]
>>> a.pop()
3
>>> a
[7, 4, 6, 8, 2]
>>> a.remove(8)
[7, 4, 6, 2]
```

None of these operations create new lists. They all update the same list.

The slicing operator here does *not* make a new list. It refers to elements and positions in the original list.

Python Lists: List Comprehensions

List comprehensions allow us to create new lists in the style of generator expressions.

```
>>> a = [2, 4, 6]
>>> b = [2*item for item in a]
```

3. Add twice the element to the new list.

2. Call each element `item`.

1. Take the list a.

```
>>> b
[4, 8, 12]
```

Python Lists: List Comprehensions

List comprehensions allow us to create new lists in the style of generator expressions.

```
>>> a = [2, 4, 6]
>>> b = [2*item for item in a if item > 3]
```

3. Add twice the element to the new list…

2. Call each element `item`.

1. Take the list a.

4. … but only if `item` is greater than 3.

```
>>> b
[8, 12]
```

CONCLUSION

- Inheritance and polymorphism are two key ideas in OOP.
  - Inheritance allows us to establishes relationships (and reuse code) between two similar data types.
  - Polymorphism allows functions to work on many types of data.
- Everything in Python is an object.
- Python OOP allows us to override standard operators.
- Python has built-in mutable lists.
- ***Preview***: More about lists and dictionaries.