Object-oriented paradigm

Object-oriented paradigm

- In conventional programming, data and computation are defined separately.
- In object-oriented programming, data and computation must be defined together.

What is an object?

In a nut shell, an object is data equipped with relevant computation. Formally, an *object* is defined by:

- A set of attributes that define data.
- A set of operations (or behaviors) that define actions on data.

What is an object?

Example. Lucy, a dog, is an object defined by the following:

- Attributes: female, Golden Retriever, 22 in., 60 lbs., 5 years old.
- Operations (or behaviors): bark, walk, run, sit, jump, swim, ...

What is an object?

Example. Rusty, a dog, is an object defined by the following:

- Attributes: male, German Shepard, 25 in., 70 lbs., 7 years old.
- Operations (or behaviors): bark, walk, run, sit, jump, swim, ...

Objects

Many objects share common attribute types and operations. For example, all dogs share:

- Attributes: gender, breed, height, weight, age.
- Operations (or behaviors): bark, walk, run, sit, jump, swim, ...

In fact, all dog objects can be defined by these attributes and operations.

What is a class?

In a nut shell, a class is an abstract definition of objects that share common attribute types and operations. Formally, a *class* is defined by:

- A set of attributes without specific data.
- A set of operations (or behaviors) that define actions on data.

Classes and objects

- A class is an abstract description of objects that share common attribute types and operations.
- A class may be regarded as simply collection of objects that share common attribute types and operations.
- An object is an instance of a class. For example, the object Lucy is an instance of the class Dog.

Classes and objects

- In Python, an object must be defined as an instance of a class.
- Every Python class has a *constructor* that instantiates an object.
- Attributes are often called *fields*, and operations (or behaviors) are called *methods*.

Python class

```
class Dog(object):
```

Constructor

Methods (operations or behaviors)

Constructor

- A *constructor* is a special function that instantiates an object.
- Objects are created from a class by invoking a constructor.
- Constructors usually have parameters, and constructors create objects according to the values of parameters.

Defining a constructor

```
class Dog(object):

  def __init__(self, b = 'German Shepherd',
    g = 'male', h = 20, w = 65, a = 5):
    self.breed = b
    self.gender = g
    self.height = h
    self.weight = w
    self.age = a
```

Methods (operations or behaviors)

Methods

- *Methods* define objects' operations, similar to the way functions define basic computational tasks.
- Many (but not all) methods return values, just like many functions return values.

Defining methods

```
class Dog(object):

  def __init__(self, b = 'German Shepherd',
    g = 'male', h = 20, w = 65, a = 5):
    self.breed = b
    self.gender = g
    self.height = h
    self.weight = w
    self.age = a

  def get_older(self):
    self.age = self.age + 1

  def gain_weight(self, w):
    self.weight = self.weight + w
```


Defining methods

```
class Dog(object):

  def __init__(self, b = 'German Shepherd',
    g = 'male', h = 20, w = 65, a = 5):
    self.breed = b
    self.gender = g
    self.height = h
    self.weight = w
    self.age = a

  def older_dog(self, another_dog):
    if self.age > another_dog.age:
      return self
    else:
      return another_dog
```

Invoking methods

- Methods define objects' operations.
- To invoke a method, one must specify a subject *with a dot* before the method name.
- To invoke a method of a class `C`, a subject must be an object of the same class `C` (or a class compatible with `C`).

Invoking methods

Example. Consider

```
class C(object):

  def method(self, p1, ...):
    ...
```

The parameter `self` refers the subject of a method invocation.

```
>>> c.method(a1, ...)
```


Invoking methods

```
>>> rusty = Dog()
>>> lucy = Dog('Golden Retriever',
    'female', 22, 60, 6)
>>> print rusty.age
5
>>> print lucy.age
6
>>> smart_dog = lucy.older_dog(rusty)
>>> print smart_dog.breed
Golden Retriever
```

The method __str__

Every class should have a method to convert an object to a string.

```
class Time(object):

  def __str__(self):
    return '%.2d:%.2d:%.2d' % \
      (self.hour, self.minute, self.second)
```

To print an object,

```
>>> time = Time(1, 7, 54)
>>> print time
01:07:54
```


Operator overloading

Python lets you define arithmetic operators (such as +) on objects. For example,

```
class Time(object):

  def __add__(self, another_time):
    sec = self.to_seconds() + another_time.to_seconds()
    sum = Time()
    return sum.from_seconds(sec)
```

To add `Time` objects,

```
>>> time = Time(9, 45, 12)
>>> duration = Time(1, 35, 56)
>>> print start + duration
11:21:08
```


Standalone functions

- A method must be defined in a class.
- Sometimes it makes sense to define an operation independent of any class.
- Such an independent operation should be defined as a standalone function.

Standalone function

Example. If we wish to compare the weight of a dog and the weight of a cat, it makes sense to define a standalone function.

```
def weight_difference(dog, cat):
  d = dog.weight - cat.weight
  if d < 0: return -d
  else return d
```


Invoking standalone functions

```
>>> rusty = Dog()
>>> abe = Cat()
>>> print weight_difference(rusty, abe)
20
```


Inheritance

Dogs and police dogs

We defined a class of dogs all of which share:

- Attributes: gender, breed, height, weight, age.
- Operations (or behaviors): bark, walk, run, sit, jump, swim, ...

Now, we wish to create another class for police dogs.

Dogs and police dogs

- Clearly, police dogs are dogs, so police dogs have all the attributes and behaviors of regular dogs.
- However, police dogs also have additional attributes (such as unit, department, etc.) and behaviors (sniff, attack, etc.)
- Can we somehow create a police-dog class as a special case of the class `Dog`?

Inheritance

- *Inheritance* refers to the notion of one class inheriting the attributes and behaviors of another class.
- In Python, inheritance occurs when we define a class as a *subclass* of another.
- For example, a police-dog class may be defined as a subclass of the class `Dog`.

A subclass of Dog

```
class Police_Dog(Dog):
```

Police_Dog's constructor

Police_Dog's additional methods

This Police_Dog class automatically inherits all the methods from the class Dog.

A subclass of Dog

```
class Police_Dog(Dog):

  def __init__(self, b = 'German Shepherd',
    g = 'male', h = 20, w = 65, a = 5, u = 1,
    d = 'Hartford'):
    self.breed = b
    self.gender = g
    self.height = h
    self.weight = w
    self.age = a
    self.unit = 1
    self.department = 'Hartford'
```

Police_Dog's additional methods

A subclass of Dog

```
class Police_Dog(Dog):

  def __init__(self, b = 'German Shepherd',
    g = 'male', h = 20, w = 65, a = 5, u = 1,
    d = 'Hartford'):
      .
      .
      .
    self.unit = 1
    self.department = 'Hartford'

  def attack(self):
    print 'WWWWOOOOFFFF!!!!'
```

Invoking methods

```
>>> eddie = Police_Dog()
>>> print eddie.weight
65
>>> eddie.gain_weight(5)
>>> print eddie.weight
70
>>> eddie.attack()
WWWWOOOOFFFF!!!!
>>> lucy = Dog()
>>> lucy.attack()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Dog' object has no attribute
'attack'
```