Information Infrastructure II

INFO I211 – Spring 2014 – Sections 18530 & 22519

Lecture 8 – 2014.02.10 & 2014.02.11

Instructor:
Mitja Hmeljak,
http://mypage.iu.edu/~mitja
mitja@indiana.edu

Group Task:

Design a class to keep information about bicycles

In the object, remember:

- the type (street, racing, mountain, etc.)
- the brand
- the price (make the price semi-private)

In your class code, remember the total number of bicycles

Provide a class (static) method to find the number of bicycles

Provide object methods to find out title, author, & price and to print the bicycle's attributes

Instantiate 3 bicycles and print the number of bicycles and each of the bicycles

Print the title, author, and price of one bicycle using the object methods you designed to obtain these individual attributes

Understanding Object Encapsulation

object *client* code should

- Communicate with objects through method *parameters* and *return values*
- *Avoid* directly altering value of an object's attribute

Objects should

- Update their *own* attributes
- Keep themselves safe by providing only *indirect* access to attributes through methods

Using Private Attributes and Private Methods

Public: Can be directly accessed by client code

Private: Cannot be directly accessed (easily) by client code

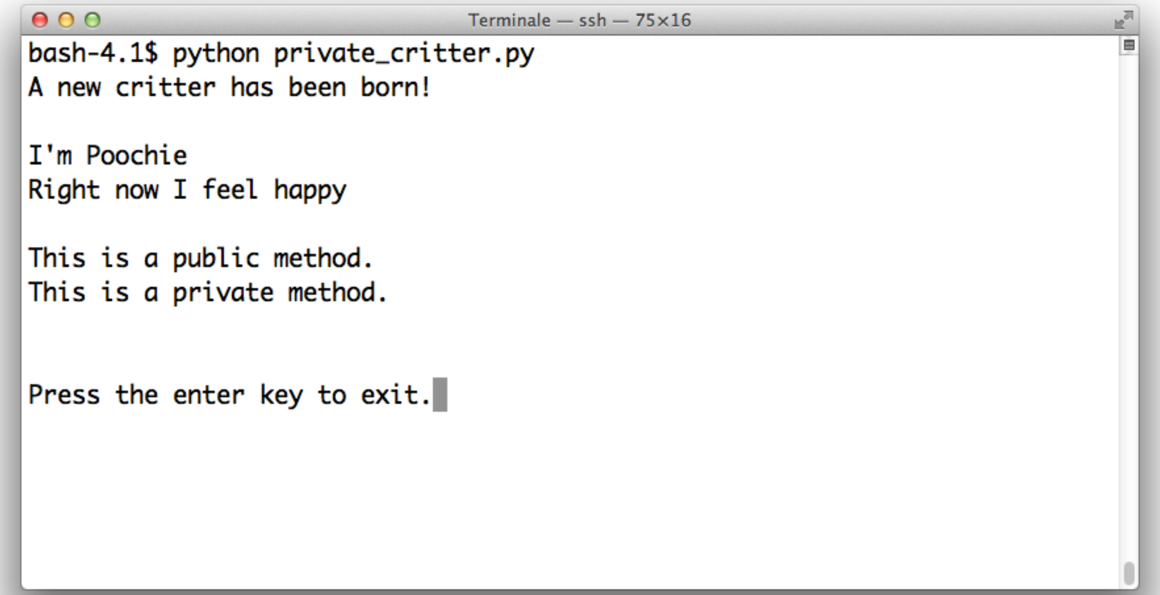
Public attribute or ***method*** can be accessed by client code

Private attribute or ***method*** cannot be (easily) accessed by client code

By default, all attributes and methods are *public*

But, we can define an attribute or method as *private*

The Private Critter Program



```
bash-4.1$ python private_critter.py
A new critter has been born!

I'm Poochie
Right now I feel happy

This is a public method.
This is a private method.


Press the enter key to exit.
```

Figure 8.8: Sample run of the Private Critter program

Object's Private attribute and private method are indirectly accessed.

Creating Private Attributes

```
class Critter(object):
    def __init__(self, name, mood):
        self.name = name    # public attribute
        self.__mood = mood # private attribute
```

name

- Created same as any attribute before
- Public attribute (default)

__mood

- Private attribute
- Two underscore characters make attribute private
- Begin any attribute with two underscores to make private

Accessing Private Attributes

```
class Critter(object):
...
    def talk(self):
        print "\nI'm", self.name
        print "Right now I feel", self.__mood, "\n"
```

Private attributes

- Can be accessed inside the class
- Can't be accessed directly through object
    - `crit1.__mood` won't work
- Technically possible to access through object, but shouldn't

Creating Private Methods

```
class Critter(object):
...
    def __private_method(self):
        print "This is a private method."
```

Like private attributes, private methods defined by two leading underscores in name

`__private_method()` is a private method

Accessing Private Methods

```
class Critter(object):
...
    def public_method(self):
        print "This is a public method."
        self.__private_method()
```

Like private attributes, private methods:

- Can be accessed inside class
- Can't be accessed directly through object
  - `crit1.__private_method()` won't work
- Technically possible to access through object, but shouldn't

Respecting an Object's Privacy

```
crit = Critter(name = "Poochie", mood = "happy")
crit.talk()
crit.public_method()
```

This code accesses only public methods

Public methods, because they belong to the class, *can* access private methods and attributes

private_critter.py

semi-private_critter.py

Understanding When to Implement and Respect Privacy

Classes

- *Write methods,* to avoid the need to directly access object's attributes
- Use privacy only for attributes and methods that are *completely internal* to operation of object

Objects

- Minimize direct reading of object's attributes
- Avoid directly altering object's attributes
- Never directly access object's private attributes or methods

Understanding *New-Style* and *Old-Style* Classes

```
class Critter(object):    # new-style class
class Critter:            # old-style class
```

New-style class: A class that is directly or indirectly based on the built-in object

Old-style class: A class that is not based on object, directly or indirectly

New-style classes

- Introduced in Python 2.2
- Significant improvements over old-style
- Create instead of old-style classes whenever possible

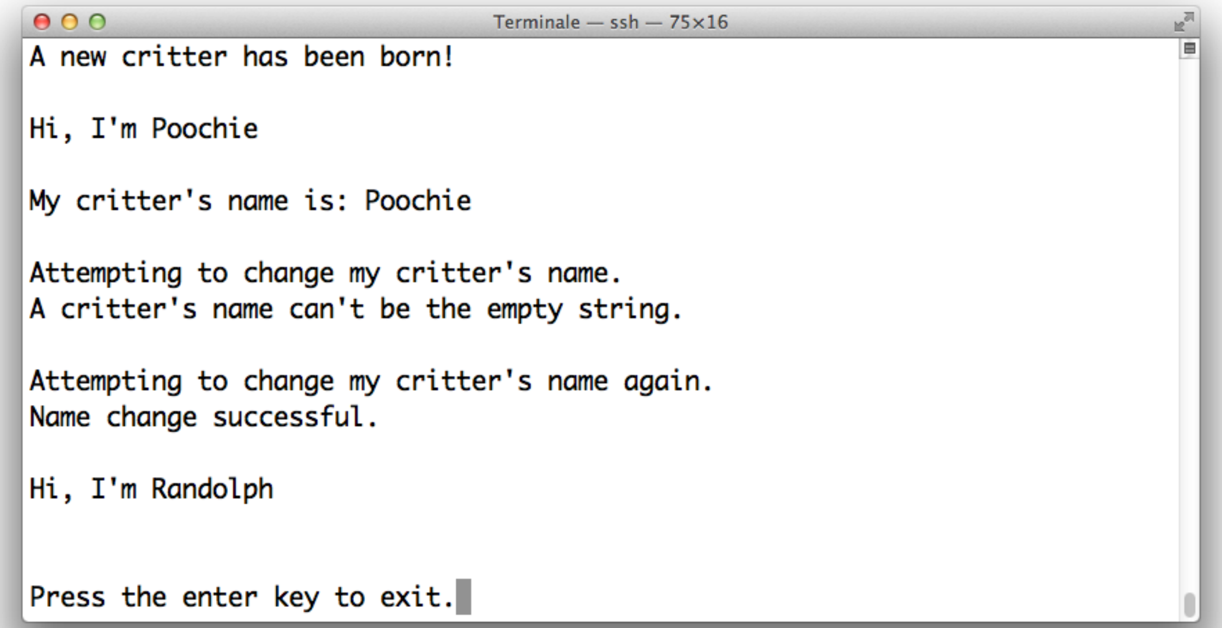
Controlling Attribute Access

Instead of denying access to an attribute, can limit access to it

Example: client code can *read*, but not *change* attribute

Properties can manage how attribute is accessed or changed

The Property Critter



```
A new critter has been born!

Hi, I'm Poochie

My critter's name is: Poochie

Attempting to change my critter's name.
A critter's name can't be the empty string.

Attempting to change my critter's name again.
Name change successful.

Hi, I'm Randolph


Press the enter key to exit.
```

Figure 8.9: Sample run of the Property Critter program

Property controls access to Critter object's attribute for its name.

Using Get Methods

```
class Critter(object):
...
    def get_name(self):
        return self.__name
```

Get method: A method that gets the value of an attribute, which is often private; by convention, name starts with "get"

get_name() provides indirect access to __name

Using Get Methods (continued)

```
>>> crit = Critter("Poochie")
>>> print crit.get_name()
Poochie
```

get_name() returns string for Critter object's name

Using Set Methods

```
class Critter(object):
...
  def set_name(self, new_name):
    if new_name == "":
      print "Critter's name can't be empty string."
    else:
      self.__name = new_name
      print "Name change successful."

  name = property(get_name, set_name)
```

Using Set Methods (continued)

```
>>> crit.set_name("")
Critter's name can't be empty string.
>>> crit.set_name("Randolph")
Name change successful.
>>> print crit.get_name()
Randolph
```

Set method: Sets an attribute, often private, to a value; by convention, name starts with "set"

set_name() allows a value to be assigned to private variable __name; imposes restriction that the value cannot be the empty string

Using Properties

```
class Critter(object):
...
    name = property(get_name, set_name)
```

- **Property:** An *interface* that allows indirect access to an attribute by wrapping access methods around dot notation
- property() function
  - Takes accessor methods and returns a property
  - Supply with *get* and *set* methods for controlled access to private attribute
  - Supply only *get* method for "read-only" property

Using Properties (continued)

```
>>> print crit.name
Randolph
>>> crit.name = "Sammy"
Name change successful.
>>> print crit.name
Sammy
>>> crit.name = ""
Critter's name can't be empty string.
```

property_critter.py critter_caretaker.py

Group Task:

Design a class to keep information about bicycles

In the object, remember:

- the type (street, racing, mountain, etc.)
- the brand
- the price (make the price semi-private)

In your class code, remember the total number of bicycles

Provide a class (static) method to find the number of bicycles

Provide object methods to find out title, author, & price and to print the bicycle's attributes

Instantiate 3 bicycles and print the number of bicycles and each of the bicycles

Print the title, author, and price of one bicycle using the object methods you designed to obtain these individual attributes

Summary

Object-oriented Programming (OOP) is a methodology of **programming** where new types of **objects** are defined

An **object** is a **single software unit** that combines *attributes* and *methods*

An **attribute** is a "*characteristic*" of an object; it's a variable associated with an object ("instance variable")

A **method** is a "*behavior*" of an object; it's a function associated with an object

A **class** defines the *attributes* and *methods* of a kind of *object*

Summary (continued)

Each **instance** method must have a special first parameter, called self by convention, which provides a way for a method to refer to object itself

A **constructor** is a special method that is automatically invoked right after a new object is created

A **class attribute** is a single attribute that's associated with a class itself

A **static method** is a method that's associated with a class itself

Summary (continued)

Public attributes and **methods** can be directly accessed by client code

Private attributes and **methods** cannot (easily) be directly accessed by client code

A **get method** gets the value of an attribute; by convention, its name starts with "get"

A **set method** sets an attribute to a value; by convention, its name starts with "set"

A **property** wraps access (**get** and **set**) methods around dot notation syntax