CS 61A
Discussion 6

Inheritance and Nonlocal

Raymond Chan
Discussion 121
UC Berkeley

Agenda

- Announcements
- Object Oriented Programming
- Inheritance
- Nonlocal

Announcements

- HW 4 due Wednesday 3/9
- Ants Project (to be released soon) due Thursday 3/17
    - Midterm 2 Wednesday 3/30 (after spring break)
- CSM Adjunct Sections sign-ups available again
    - http://csmscheduler.herokuapp.com/

Object Oriented Programming

- Treat data as objects (like real life).
- We can mutate an object's data rather than recreate it.
- A class serve as a template for creating objects.

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

Object Oriented Programming

- To create an object from the class, we need to create an instance of a class.

- Initializing an instance calls the __init__ method.

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

>>> buddy = Dog(“Buddy”, “Gold”)

Object Oriented Programming

- Every dog has certain details but are unique to the dog.
- These are instance attributes (**name, color**)

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy = Dog("Buddy", "Gold")
>>> molly = Dog("Molly", "White")
>>> buddy.name
"Buddy"
>>> molly.color
"White"
```

Object Oriented Programming

- Attributes that shared among all instance are class attributes (**num_legs**)

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy = Dog("Buddy", "Gold")
>>> molly = Dog("Molly", "White")
>>> buddy.num_legs
4
>>> molly.num_legs
4
```

Object Oriented Programming

- We can have instances have attributes that override the class attribute

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy.num_legs = 5
>>> buddy.num_legs
5
>>> Dog.num_legs
4
>>> molly.num_legs
4
```

Object Oriented Programming

- Objects have actions or functions that belong to the object.
- Methods are functions that all instances can perform

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy.eat("food")
Buddy ate a food
>>> molly.eat("candy")
Molly ate a candy
```

Object Oriented Programming

- The **self** argument is passed in implicitly if you call the method via the instance.

- We can also call it from the class, but we must pass in the instance.

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy.eat("food")
Buddy ate a food
>>> Dog.eat(buddy, "food")
Buddy ate a food
>>> Dog.eat("stuff")
Error
```

Object Oriented Programming

- For any instance, we can define methods after we've created the object.

- But it is only defined for that specific instance.

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy = Dog("Buddy", "Gold")
>>> molly = Dog("Molly", "White")
>>> buddy.f = lambda x: x*x
>>> buddy.f(2)
4
>>> molly.f(5)
Error
```

```
>>> Dog.f(5)
Error
```

Object Oriented Programming

- We can also define a method function for the class.

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy = Dog("Buddy", "Gold")
>>> molly = Dog("Molly", "White")
>>> Dog.f = lambda self, x: self.num_legs*x
>>> buddy.f(2)
8
>>> molly.f(5)
20
```

```
>>> Dog.f(buddy, 5)
20
```

Object Oriented Programming

- Notice that if you define the method function for the class, you need to have ***self*** as the first parameter.

- Thus you can access an instance via ***self***.

- This cannot be done if you define a method via an instance.

```
class Dog(object):
    num_legs = 4

    def __init__(self, name, color):
        self.name = name
        self.color = color

    def eat(self, thing):
        print(self.name + " ate a " + str(thing))
```

```
>>> buddy = Dog("Buddy", "Gold")
>>> molly = Dog("Molly", "White")
>>> Dog.f = lambda self, x: self.num_legs*x
>>> buddy.f(2)
8
>>> molly.f = lambda self, x: self.num_legs+x
Error: missing argument x
```

OOP Q1

```
class Instructor:
    degree = "PhD"
    def __init__(self, name):
        self.name = name

    def lecture(self, topic):
        print("Today we're learning about " + topic)

hilfinger = Instructor("Professor Hilfinger")
```

```
class TeachingAssistant:
    def __init__(self, name):
        self.name = name
        self.students = {}

    def add_student(self, student):
        self.students[student.name] = student

    def assist(self, student):
        student.understanding += 1
```

```
class Student:
    instructor = hilfinger

    def __init__(self, name, ta):

        self.name = name
        self.understanding = 0
        ta.add_student(self)

    def attend_lecture(self, topic):
        self.instructor.lecture(topic)
        print(Student.instructor.name + " is awesome!")
        self.understanding += 1

    def visit_office_hours(self, staff):
        staff.assist(self)
        print("Thanks, " + staff.name)
```

OOP Q1

```
>>> soumik = TeachingAssistant("Soumik")
>>> kelly = Student("Kelly", soumik)
>>> kelly.attend_lecture("OOP")
```

OOP Q1

```
>>> soumik = TeachingAssistant("Soumik")
>>> kelly = Student("Kelly", soumik)
>>> kelly.attend_lecture("OOP")
Today we're learning about OOP
Professor Hilfinger is awesome!
```

OOP Q1

```
>>> kristin = Student("Kristin", soumik)
>>> kristin.attend_lecture("trees")
```

OOP Q1

```
>>> kristin = Student("Kristin", soumik)
>>> kristin.attend_lecture("trees")
Today we're learning about trees
Professor Hilfinger is awesome!
```

OOP Q1

```
>>> kristin.visit_office_hours(TeachingAssistant("James"))
```

OOP Q1

```
>>> kristin.visit_office_hours(TeachingAssistant("James"))
Thanks, James
```

OOP Q1

```
>>> kristin.visit_office_hours(TeachingAssistant("James"))
Thanks, James
>>> kelly.understanding
```

OOP Q1

```
>>> kristin.visit_office_hours(TeachingAssistant("James"))
Thanks, James
>>> kelly.understanding
1
```

OOP Q1

```
>>> kristin.visit_office_hours(TeachingAssistant("James"))
Thanks, James
>>> kelly.understanding
1
>>> soumik.students["Kristin"].understanding
```

OOP Q1

```
>>> kristin.visit_office_hours(TeachingAssistant("James"))
Thanks, James
>>> kelly.understanding
1
>>> soumik.students["Kristin"].understanding
2
```

OOP Q1

```
>>> Student.instructor = Instructor("Professor DeNero")
>>> Student.attend_lecture(kelly, "lists")
```

OOP Q1

```
>>> Student.instructor = Instructor("Professor DeNero")
>>> Student.attend_lecture(kelly, "lists")
Today we're learning about lists
Professor DeNero is awesome!
```

Inheritance

```
class Dog(object):
    def __init__(self, name, owner, color):
        self.name = name
        self.owner = owner
        self.color = color
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")

class Cat(object):
    def __init__(self, name, owner, lives=9):
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")
```

Inheritance

```
class Dog(object):
    def __init__(self, name, owner, color):
        self.name = name
        self.owner = owner
        self.color = color
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")

class Cat(object):
    def __init__(self, name, owner, lives=9):
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")
```

Inheritance

- Both Dog and Cat classes have do pretty much the same thing with a few specific differences.

- Rather than repeat so much code, we can use inheritance.

- A class can inherit the instance variables and methods of a another class.

Inheritance

```
class Pet(object):
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(eslf):
        print(self.name)

class Dog(Pet):
    def __init__(self, name, owner, color):
        Pet.__init__(self, name, owner)
        self.color = color
    def talk(self):
        print(self.name + " says woof!”)
```

The base class
Or Dog’s super class

The subclass

Inheritance

- A Dog is a Pet, and thus the Dog class can inherit the Pet class.
- By redefining **__init__** and **talk**, the subclass overrides the super class's methods.
- Use the super class's methods but add attributes or actions that are unique to the subclass.

```
class Dog(Pet):
   def __init__(self, name, owner, color):
      Pet.__init__(self, name, owner)
      self.color = color
   def talk(self):
      print(self.name + " says woof!”)
```

Inheritance - Cat

```
class Cat(Pet):

    def __init__(self, name, owner, lives=9):


    def talk(self):

    def lose_life(self):
```

Inheritance - Cat

```
class Cat(Pet):

    def __init__(self, name, owner, lives=9):
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
      self.lives = lives
```

Inheritance - Cat

```
class Cat(Pet):

    def __init__(self, name, owner, lives=9):
        Pet.__init__(self, name, owner)
        self.lives = lives

    def talk(self):
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
      self.lives = lives

   def talk(self):
      print(self.name + " says meow!”)
```

Inheritance - Cat

```
class Cat(Pet):

    def __init__(self, name, owner, lives=9):
        Pet.__init__(self, name, owner)
        self.lives = lives

    def talk(self):
        print(self.name + " says meow!”)

    def lose_life(self):
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
      self.lives = lives

   def talk(self):
      print(self.name + " says meow!”)

def lose_life(self):
   if self.lives > 0:



   else:
      print("No more lives.")
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
      self.lives = lives

   def talk(self):
      print(self.name + " says meow!”)

def lose_life(self):
   if self.lives > 0:
      self.lives -= 1


   else:
      print("No more lives.")
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
      self.lives = lives

   def talk(self):
      print(self.name + " says meow!”)

def lose_life(self):
   if self.lives > 0:
      self.lives -= 1
      if self.lives == 0:

   else:
      print("No more lives.")
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
      self.lives = lives

   def talk(self):
      print(self.name + " says meow!”)

def lose_life(self):
   if self.lives > 0:
      self.lives -= 1
      if self.lives == 0:
         self.is_alive = False
   else:
      print("No more lives.")
```

Inheritance - Cat

```
class Cat(Pet):

   def __init__(self, name, owner, lives=9):
      Pet.__init__(self, name, owner)
      self.lives = lives

   def talk(self):
      print(self.name + " says meow!”)

   def lose_life(self):
      if self.lives > 0:
         self.lives -= 1
         if self.lives == 0:
            self.is_alive = False
      else:
         print("No more lives.")
```

Only this instance’s is_alive is False

Nonlocal

- We could only access variables in parent frames and not modify them.
- **Nonlocal** allows us to modify variables in parents frame and outside of the current frame.

Nonlocal

- We could only access variables in parent frames and not modify them.
- **Nonlocal** allows us to modify variables in parents frame and outside of the current frame.

```
def stepper(num):
  def step():

    num = num + 1
    return num
  return step
```

Error: We are trying to use **num** before we assigned it

Nonlocal

- We could only access variables in parent frames and not modify them.
- **Nonlocal** allows us to modify variables in parents frame and outside of the current frame.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
```

Nonlocal

- We could only access variables in parent frames and not modify them.
- **Nonlocal** allows us to modify variables in parents frame and outside of the current frame.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
```

For environment diagrams, **num** is not a variable in any frame labeled **step**

Nonlocal

```
a = 5
def another_add_one():
   nonlocal a
   a += 1
another_add_one()
```

Nonlocal

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

Nonlocal cannot be used to modify variables in the global frame.

Nonlocal

```
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

Nonlocal

```
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

y does not exist in any parent frames.
It is a local variable

Nonlocal

```
def adder(x):
    z = 5
    def add(y):
        z = 8
        nonlocal x, z
        x += z
        return x
    return add
adder(2)(3)
```

Nonlocal

```
def adder(x):
    z = 5
    def add(y):
        z = 8
        nonlocal x, z
        x += z
        return x
    return add
adder(2)(3)
```

z is defined before nonlocal

Nonlocal

- Global variables cannot be modified using the nonlocal keyword.
- Variables in the current frame cannot be overridden using the nonlocal keyword.

Recap

- OOP allows use to treat data as objects.
- Class serves as a template for instance objects.
- Use inheritance to avoid repeating code on if there is a **“is-a”** relationship between the two classes.
- Nonlocal allows us to modify variables in the parent frame.