CS 61A
DISCUSSION 5

OBJECT ORIENTED PROGRAMMING

Raymond Chan
Discussion 134
UC Berkeley Fall 16

AGENDA

- Announcements
- Practice Question
- O.O.P
    - Advanced section slides online.
- Inheritance
- Challenge Questions

ANNOUNCEMENTS

- Ants due 10/14 (submit early for extra point)
- HW 6 due tonight (HW Party 6:30-8:30)
- HW 7 due 10/11
- Guerrilla Section on objects and growth
- 1 on 1 tutoring
- Lab 6 due Friday

CHALLENGE QUESTION 1

2.3 SUMMER 2013 FINAL

```
class A:
    def f(self):
        return 2
    def g(self, obj, x):
        if x == 0:
            return A.f(obj)
        return obj.f() + self.g(self, x - 1)

class B(A):
    def f(self):
        return 4

>>> x, y = A(), B()
>>> x.f()

>>> B.f()

>>> x.g(x, 1)

>>> y.g(x, 2)
```

CHALLENGE QUESTION 2

2.3 SUMMER 2013 FINAL

- Implement the **Yolo** class so that the following interpreter session works as expected (Summer 2013 Final).

```
>>> x = Yolo(1)
>>> x.g(3)
4
>>> x.g(5)
6
>>> x.motto = 5
>>> x.g(5)
10
```

PRACTICE QUESTION

(d) (1.5 pt) Consider the following function for computing powers of a polynomial:

```
def polypow(P, k):
    """P ** k, where P is a polynomial and K is a
    non-negative integer."""
    result = Poly(1)
    while k != 0:
        if k % 2 == 1:
            result = result.mult(P)
        P = P.mult(P)
        k = k // 2
```

Circle the order of growth that best describes the worst-case execution time of `polypow`, as a function of k, where execution time is measured in the number of times that the `.mult` method is called.

A. $\Theta(k)$

B. $\Theta(k^2)$

C. $\Theta(\sqrt{k})$

D. $\Theta(\log k)$

E. $\Theta(2^k)$

PRACTICE QUESTION

(d) (1.5 pt) Consider the following function for computing powers of a polynomial:

```
def polypow(P, k):
    """P ** k, where P is a polynomial and K is a
    non-negative integer."""
    result = Poly(1)
    while k != 0:
        if k % 2 == 1:
            result = result.mult(P)
        P = P.mult(P)
        k = k // 2
```

All operations are constant time, including **Poly(1)**.

Circle the order of growth that best describes the worst-case execution time of `polypow`, as a function of k, where execution time is measured in the number of times that the `.mult` method is called.

A. $\Theta(k)$

B. $\Theta(k^2)$

C. $\Theta(\sqrt{k})$

D. $\Theta(\log k)$

E. $\Theta(2^k)$

Iterate until **k** reaches 0.
In each iteration, whether **k** is odd or even,
we call **mult**, a constant time operation.
k is *reduced by half* at each step.
Thus **Θ(log k)**

OBJECT ORIENTED PROGRAMMING

- Treat data as objects (like real life).
- We can mutate an object's data rather than recreate it.
- A ***class*** serves 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")
>>> molly = Dog("Molly", "White")
>>> buddy.name
"Buddy"
>>> molly.color
"White"
```

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

- Remember to set ***instance attributes*** in the **__init__** class.
- Otherwise the arguments passed in would be lost.
- ***Instance attributes*** can be set in other functions too.

```
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

- 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

- Instances can have an instance attribute 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 = Dog("Buddy", "Gold")
>>> molly = Dog("Molly", "White")
>>> buddy.num_legs = 5
>>> buddy.num_legs
5
>>> Dog.num_legs
4
>>> molly.num_legs
4
```

OBJECT ORIENTED PROGRAMMING

- Objects have actions that belong to the object.
- Bound methods are functions that all instances can call.

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

OBJECT ORIENTED PROGRAMMING

- The **self** argument is passed in implicitly if you invoke the method via the instance.
- We can also call it from the class, but we must pass in 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")
>>> buddy.eat("food")
Buddy ate a food
>>> Dog.eat(buddy, "food")
Buddy ate a food
>>> Dog.eat("stuff")
Error (not enough arguments)
```

OBJECT ORIENTED PROGRAMMING

- A function instead a class without **self** as the first parameter is simply a function.
- Cannot access class and instance attributes.

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

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

    def eat(thing):
        print("ate a ", thing)

    def eat1(thing):
        print(num_legs, "ate a", thing)
```

```
>>> buddy = Dog("Buddy", "Gold")
>>> Dog.eat("food")
ate a food
>>> buddy.eat("candy")
Error (too many arguments)
>>> Dog.eat1("candy")
Error (no global variable num_legs)
```

OBJECT ORIENTED PROGRAMMING

Q1

```
class Student:
    instructor = dumbledore

    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)
```

```
class Instructor:
    degree = “PhD (Magic)”
    def __init__(self, name):
        self.name = name

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

dumbledore = Instructor("Dumbledore")
```

```
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
```

OBJECT ORIENTED PROGRAMMING

Q1

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")


>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")


>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))

>>> harry.understanding

>>> snape.students["Hermione"].understanding

>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
```

OBJECT ORIENTED PROGRAMMING

Q1

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")
Today we're learning about potions
Dumbledore is awesome!
>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")
```

```
>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))

>>> harry.understanding

>>> snape.students["Hermione"].understanding

>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
```

OBJECT ORIENTED PROGRAMMING

Q1

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")
Today we're learning about potions
Dumbledore is awesome!
>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")
Today we're learning about herbology
Dumbledore is awesome!
>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))

>>> harry.understanding

>>> snape.students["Hermione"].understanding

>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
```

OBJECT ORIENTED PROGRAMMING

Q1

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")
Today we're learning about potions
Dumbledore is awesome!
>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")
Today we're learning about herbology
Dumbledore is awesome!
>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))
Thanks, Hagrid
>>> harry.understanding

>>> snape.students["Hermione"].understanding

>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
```

OBJECT ORIENTED PROGRAMMING

Q1

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")
Today we're learning about potions
Dumbledore is awesome!
>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")
Today we're learning about herbology
Dumbledore is awesome!
>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))
Thanks, Hagrid
>>> harry.understanding
1
>>> snape.students["Hermione"].understanding

>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
```

OBJECT ORIENTED PROGRAMMING

Q1

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")
Today we're learning about potions
Dumbledore is awesome!
>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")
Today we're learning about herbology
Dumbledore is awesome!
>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))
Thanks, Hagrid
>>> harry.understanding
1
>>> snape.students["Hermione"].understanding
2
>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
```

OBJECT ORIENTED PROGRAMMING

Q1

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")
Today we're learning about potions
Dumbledore is awesome!
>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")
Today we're learning about herbology
Dumbledore is awesome!
>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))
Thanks, Hagrid
>>> harry.understanding
1
>>> snape.students["Hermione"].understanding
2
>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
Today we're learning about transfiguration
Umbridge is awesome!
```

Since the class attribute changed, the instance accessed the new instructor instance.

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 do pretty much the same thing with a few specific differences.
- Rather than repeat so much code, we can make use of **inheritance**.
- A class can ***inherit*** the class attributes, instance attributes, and methods of a another class.

INHERITANCE

- The **Bar** class inherits from the **Foo** class.
- **Foo** is the ***base class.***
  - inheriting from
- **Bar** is the ***sub class.***
  - does the inheriting
- By default Python objects inherits from the **object** class.

```
class Foo(object):

class Bar(Foo):
```

INHERITANCE

- The **Bar** class inherits from the **Foo** class.
- **Foo** is the ***base class.***
  - inheriting from
- **Bar** is the ***sub class.***
  - does the inheriting

```
class Foo(object):

class Bar(Foo):
```

INHERITANCE

- The **Bar** class inherits from the **Foo** class.
- **Foo** is the ***base class.***
  - inheriting from
- **Bar** is the ***sub class.***
  - does the inheriting
- By default Python objects inherits from the **object** class.

```
class Foo():

class Bar(Foo):
```

INHERITANCE

- Inheritance make use of a **is-a** hierarchical relationship.

```
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(self):
        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!")
```

INHERITANCE

- A Dog is a Pet, and thus the Dog class can inherit the Pet class.

```
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(self):
        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!")
```

INHERITANCE

- By redefining **__init__** and **talk**, the subclass *overrides* the base class's methods.

```
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(self):
        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!")
```

INHERITANCE

- The **Dog** class's **__init__** uses the base class's **__init__**.

```
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(self):
        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!")
```

INHERITANCE

- Uses the base class's methods but adds attributes (**self.color**) and/or actions that are unique to the subclass.

```
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(self):
        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!")
```

INHERITANCE

2.1 CAT

```
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(self):
        print(self.name)
```

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



    def talk(self):


    def lose_life(self):
```

INHERITANCE

2.1 CAT

```
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(self):
        print(self.name)
```

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

    def talk(self):


    def lose_life(self):
```

Make use of the base class's **__init__**.

A **Cat** is different from a **Pet** because it has multiple **lives**.

Add **self.lives** instance attribute.

INHERITANCE

2.1 CAT

```
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(self):
        print(self.name)
```

```
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

2.1 CAT

```
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(self):
        print(self.name)
```

```
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.")
```

Since the base class has an instance attribute of **self.is_alive**, we need to set the **Cat's self.is_alive** to False.

INHERITANCE

2.2 NOISY CAT

```
class NoisyCat(Cat):
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):



    def talk(self):
```

INHERITANCE

2.2 NOISY CAT

```
class NoisyCat(Cat):
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        Cat.__init__(self, name, owner, lives)

    def talk(self):
```

We don't actually need an **__init__**.

Since **NoisyCat** inherits from **Cat**, any new instance will call **Cat**'s **__init__**.

We are not doing anything new either.

INHERITANCE

2.2 NOISY CAT

```
class NoisyCat(Cat):
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?
        Cat.__init__(self, name, owner, lives)

    def talk(self):
        Cat.talk(self)
        Cat.talk(self)
```

Make use of the base class's method by calling it twice.

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.