CS1 Lecture 27 Mar. 27, 2017

- HW6 due Wed.
  - For question 2
    - Ensure legal moves – i.e. if user enters an illegal choice, print something appropriate and ask for a new choice.
    - Computer gameplay can be random (but must be legal). You can use, for instance, random.randint(…) for choosing number of balls. It's certainly not required, but you can make computer smarter than random if you want to.
- Discussion sections this week – object-oriented programming - will have points.

Last time

- Ch 17: classes and methods
  - Circle class similar in style to HW6 Box

Today

- Chapter 18
  - class attributes
  - inheritance

  I will not do the card example of Ch 18. I will mainly cover the two sections hamed “Class Variables” and “Inheritance” building from our initial Cat and Dog classes (catdog.py) to a more general Animal class (animals.py)

An interesting research topic:

Deniable encryption is a type of cryptography that allows an encrypted text to be decrypted in two or more ways, depending on which decryption key is used. The use of two or more keys allows the sender, theoretically, to conceal or deny the existence of a controversial message in favor of a more benign decryption. For instance, a company may send an encrypted message to its high-level administrative staff whose key decrypts the message to read "We have no plans to change our business model", while the board of directors receives the same message that using its own key decrypts the same message to read "We are going bankrupt at this rate and need to let 20,000 people go, including high-level administrators". Deniable encryption is sometimes used for misinformation purposes when the sender anticipates, or even encourages, interception of a communication.

Def from: http://searchsecurity.techtarget.com/definition/deniable-encryption

Ch 18 – Class attributes

Consider the basic Cat and Dog classes from last week. catdog.py

Each cat and dog has a name but names aren't very unique. Can we give each cat a unique ID number? **Class attributes** make this easy.

```
class Cat ():
    scientificName = 'felis catus'
```

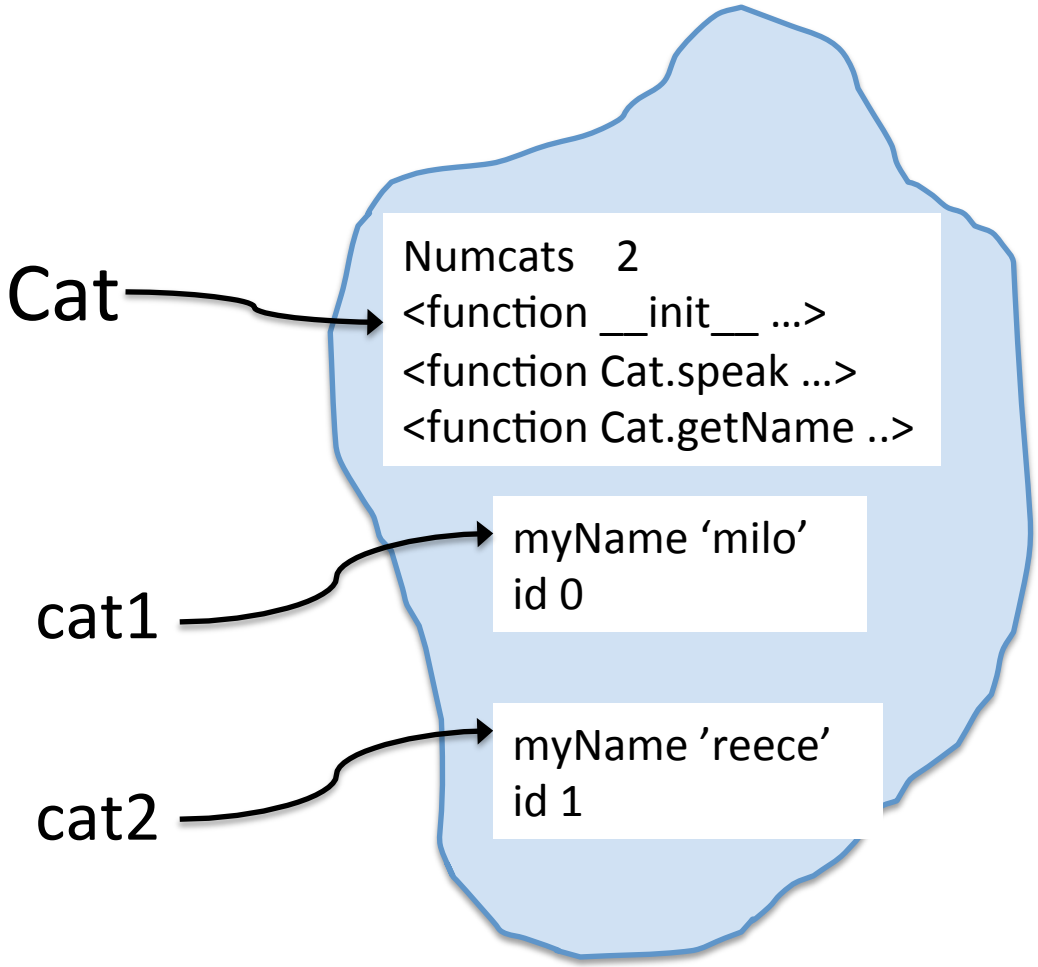
Ch 18 – Class attributes

To uniquely identify cats,

- use a class attribute, numCats, initially 0
- each time a Cat object is created
  - assign value of numCats to new Cat as id
  - increment numCats class attribute
- Also, update __repr__ to show id in printed representation

catdogClassAttrib.py

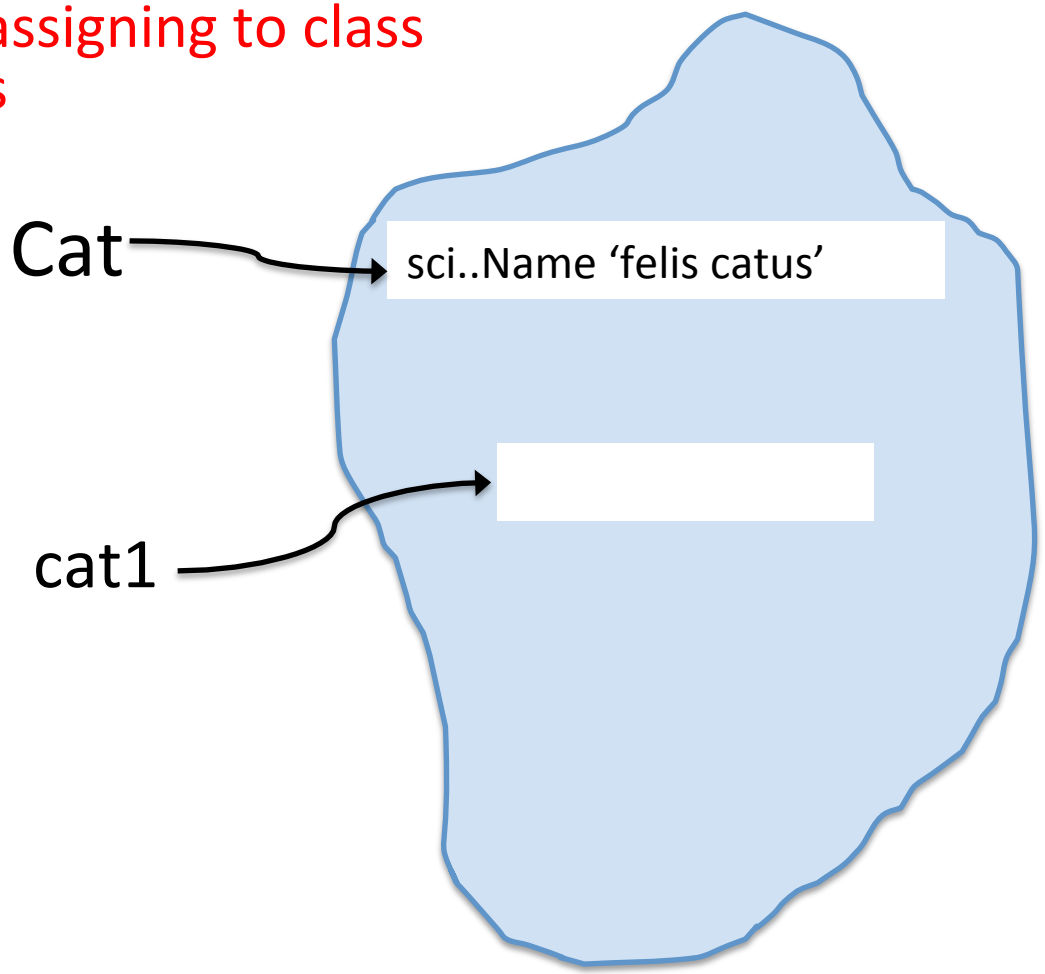
```
>>> class Cat():
        numCats = 0
        def __init__(…):
            …
        def speak(…):
            …
        …
>>> cat1 = Cat('milo')
>>> cat2 = Cat('reece')
```



Ch 18 – Be careful when assigning to class attributes

```
class Cat ():
        scientificName = 'felis catus'

>>> Cat.scientificName
'felis catus'
>>> cat1 = Cat()
>>> cat1.scientificName
'felis catus'
>>> cat1.scientficName = 'kitty'
>>> Cat.scientificName
'felis catus'
>>> cat1.scientficName
'kitty'
```



When you reference an attribute from an instance, Python first checks if the instance contains the attribute. If not, it checks if the class has it. So, the first cat1.scientificName above yields 'felis catus'.

BUT, when you **assign** to an attribute from an instance, Python uses the namespace of the object. So cat1.scientificName does not modify the class attribute. Instead, it creates a new attribute (of the same name) in the instance.

GENERAL RULE: refer to class attributes using class name – Cat.scientificName – rather than instance

Ch 18 - Inheritance

When creating classes like Cat and Dog, some properties and methods might naturally be the same in both.

Inheritance, in object-oriented programming languages, is the ability to define a new class that is a modified version of an existing class.

```
class SubClass (SuperClass):
    …
```

It's also common to say "derived class" and "base class"

The new class SubClass **inherits** all methods (and properties) of SuperClass but can also:

- can add new methods (ones not defined in SuperClass)
- redefine (**override)** methods inherited from SuperClass

Ch 18 - Inheritance

```
>>> class Foo ():
        def doSomething(self):
            print('hi')
>>> class Bar(Foo):
        def doSomething2(self):
            print('bye')
>>> b = Bar()
>>> b.doSomething()
hi
>>> b.doSomething2()
Bye
>>> f = Foo()
>>> f.doSomething()
hi
>>> f.doSomething2()
```
Error

Ch 18 - Inheritance

```
>>> class Foo ():
        def doSomething(self):
            print('hi')
>>> class Bar(Foo):
        def doSomething(self):
            print('hello')

        def doSomething2(self):
            print('bye')
>>> b = Bar()
>>> b.doSomething()
hello
>>> f = Foo()
>>> f.doSomething()
hi
```

Ch 18 - Inheritance

```
>>> class Foo ():
        def __init__(self):
            self.x = 0

>>> class Bar(Foo):
        def __init__(self):
            self.y = 0

>>> b = Bar()
>>> b.y
0
>>> b.x
Error
```

Can we inherit the properties of the superclass Foo? Yes, by calling superclass' __init__. Not required BUT highly recommended/best practice is for __init__ of subclass to always first call __init__ of superclass.

Ch 18 - Inheritance

```
>>> class Foo ():
        def __init__(self):
            self.x = 0

>>> class Bar(Foo):
        def __init__(self):
            Foo.__init__(self)
            self.y = 0

>>> b = Bar()
>>> b.y
0
>>> b.x
0
```

Highly recommended/best practice is for __init__ of subclass to always first call __init__ of superclass.

Inheritance demo

Define Dog and Cat as subclasses of new Animal class. animals.py

Next topic

Ch 21 – Analysis of algorithms

Ch 21 Introduction to Analysis of Algorithms (or "Computational Complexity")

- The first step of programming (before implementation/ typing in the code) should be design. At this stage, you need to worry both about designing a correct solution *and* a solution that will be efficient enough (in a big picture sense) to meet your needs
- Then, after first correctly implementing your program there can be, if necessary, a *tuning* process to increase program speed as much as possible. But improvements at this stage are usually much less substantial than those created by selecting the right solution approach at design time. Sometimes, you will need to redesign and re-implement, using a better algorithm, if tuning doesn't yield enough improvement.
- Note that a critical first consideration is *correctness*. Worrying about details of *efficiency* should generally come second.
- Over the next few lectures we'll take a brief look at formal tools and techniques for thinking about "big picture" program running time and algorithmic efficiency.

WANT TO HAVE FORMAL WAY TO TALK ABOUT RUNNING TIME/EFFICIENCY DIFFERENCES BETWEEN ALGORITHMS …

HOW SIMILAR/DIFFERENT ARE THE FUNCTIONS BELOW IN RUNING TIME/EFFICIENCY?

```
def foo1(n):
    result = 0
    for i in range(n):
        for j in range(n):
            result = result + i*j
```

```
def foo2(n):
    result = 0
    for i in range(n):
        for j in range(i):
            result = result + i*j
```

```
def foo3(n):
    result = 0
    for i in range(n):
        for j in range(n):
            if (j % 2 == 0):
                result = result + i*j
```

```
def foo4(n):
    result = 0
    for i in range(n):
        for j in range(n*n):
            result = result + i*j
```