Week 13

CSC111 — Fall 2015
(Lab 12, Homework 12)

Dominique Thiébaut
dthiebaut@smith.edu
This Week: **Two Concepts**

- **Lists of Lists**
- **Class Inheritance**
Lists of Lists

(Chapter 11—Designing with Lists and Classes)
Two Types of Lists

Useful List Operations

Two Approaches to Filtering Data

Examples
Two Types of Lists

• `someList1 = [ 1, “hello”, 6.5 ]`

• `someList2 = ( 1, “hello”, 6.5 )`
Two Types of Lists

- `someList1 = [ 1, "hello", 6.5 ]`  mutable
- `someList2 = ( 1, "hello", 6.5 )`  immutable
someList1 = [ 1, 2, 3 ]
for i in range( 20, 30 ):
    someList1.append( i )

someList2 = ( 1, 2, 3 )
for i in range( 20, 30 ):
    someList2.append( i )
someList1 = [ 1, 2, 3 ]
for i in range( 20, 30 ):
    someList1.append( i )

someList2 = ( 1, 2, 3 )
for i in range( 20, 30 ):
    someList2.append( i )
Notation

```python
someList1 = [ 1, 2, 3 ]
for i in range( 20, 30 ):
    someList1.append( i )

someList2 = ( 1, 2, 3 )
for i in range( 20, 30 ):
    someList2.append( i )
```

Tuple
Two Types of Lists

Useful List Operations

Two Approaches to Filtering Data

Examples
Useful List Operations

```
Python 3.1.1 (r311:74543, Aug 24 2009, 18:44:04)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> L = [3, 10, 3, 5, 1, -1, 0, 6]
>>> L
[3, 10, 3, 5, 1, -1, 0, 6]
>>> L.sort()
>>> L
[-1, 0, 1, 3, 3, 5, 6, 10]
>>> L.reverse()
>>> L
[10, 6, 5, 3, 3, 1, 0, -1]
>>> L[0]
10
>>> L[0:3]
[10, 6, 5]
>>> L[-3:]
[1, 0, -1]
>>> S = set(L)
>>> S
{0, 1, 3, 5, 6, 10, -1}
>>> L = list(S)
>>> L
[0, 1, 3, 5, 6, 10, -1]
```
When sorting lists:

• numbers are sorted in **increasing**, numerical order

• strings are sorted in **alphabetical** order

• lists of objects cannot be sorted without adding special methods to the classes the objects are derived from (**eq__()**, **lt__()**, **gt__()**, **le__()**, **ge__()**, **ne__()**)
Useful List Operations
Sorting Tuples

```python
Python 3.1.1 (r311:74543, Aug 24 2009, 18:44:04)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> L = [ (10, "Smith"), (1, "Amherst"), (3, "Umass"), (5, "Hampshire") ]
>>> L
[(10, 'Smith'), (1, 'Amherst'), (3, 'Umass'), (5, 'Hampshire')]
>>> L.sort()
>>> L
[(1, 'Amherst'), (3, 'Umass'), (5, 'Hampshire'), (10, 'Smith')]

>>> L2 = [ ("Smith", 10), ("Amherst", 1), ("Umass", 3 ), ("Hampshire", 5 ) ]
>>> L2.sort()
>>> L2
[('Amherst', 1), ('Hampshire', 5), ('Smith', 10), ('Umass', 3)]
```
Two Types of Lists

Useful List Operations

Two Approaches to Filtering Data

Examples
Examples of Filtering Problems...
The Problem at Hand

Textual Information, in easily splittable form
OPTION 1: We are only interested in the red information, and only the smaller or larger items...
The Problem at Hand

\[
\left[
\left(\begin{array}{ccc}
\text{Textual Info.}
\end{array}\right),
\left(\begin{array}{ccc}
\text{Textual Info.}
\end{array}\right),
\left(\begin{array}{ccc}
\text{Textual Info.}
\end{array}\right),
\ldots
\left(\begin{array}{ccc}
\text{Textual Info.}
\end{array}\right)
\right]
\]
The Problem at Hand

Textual Info.

\[
\begin{align*}
&[ (\text{red} , \text{gray} , \text{gray} ), \\
& (\text{red} , \text{gray} , \text{gray} ), \\
& (\text{red} , \text{gray} , \text{gray} ), \\
& \ldots \\
& (\text{red} , \text{gray} , \text{gray} ) ]
\end{align*}
\]

SORT

\[
\begin{align*}
&[ (\text{red} , \text{gray} , \text{gray} ), \\
& (\text{red} , \text{gray} , \text{gray} ), \\
& (\text{red} , \text{gray} , \text{gray} ), \\
& \ldots \\
& (\text{red} , \text{gray} , \text{gray} ) ]
\end{align*}
\]
OPTION 2:
We are only interested in the lines that contain the red information

Textual Information, in easily splittable form
The Problem at Hand

[ ( , , ),
  ( , , ),
  ( , , ),
  ...
  ( , , ) ]

FILTER

[ ( , , ),
  ...
  ( , , ) ]
Two Types of Lists

Useful List Operations

Two Approaches to Filtering Data

Examples
Example 1
10 **Rainiest** Months In **Cambridge**, U.K.?

http://cs.smith.edu/~dthiebaut/UKTemperatures/
# rainyCambridge.py
# D. Thiebaut
# Getting started…

```python
import urllib.request

#--- some constants used by the program ---
BASEURL = "http://cs.smith.edu/~dthiebaut/UKTemperatures/


def getTemperatureData( fileName ):
    url = BASEURL + fileName
    print("Retrieving data from", url)
    f = urllib.request.urlopen( url )
    data = f.read().decode("utf-8")
    return data

def main():
    data = getTemperatureData("cambridgedata.txt")
    print( data )

main()
```
We stopped here last time...
Example 2
Ammie@hampshire.edu
Bessie@smith.edu
Carylon@smith.edu
Cheryll@smith.edu
Cordelia@smith.edu
Illa@smith.edu
Lisbeth@smith.edu
Mackenzie@smith.edu
Maryellen@smith.edu
Matha@smith.edu
Patrica@hampshire.edu
Sanjuana@smith.edu
Sharie@smith.edu
Sonya@smith.edu
Yuko@smith.edu

Cheryll@smith.edu
Codi@smith.edu
Cordelia@smith.edu
Elenore@smith.edu
Emelia@smith.edu
Josie@smith.edu
...

List of email addresses for students enrolled in several classes.

Need a list of all Smith students without duplicates and a list of all 5-College students without duplicates
Example 3
U.S. Presidents

   ...

Filter out Fields 2, 6, and 7

1. George Washington, 30/04/1789, 4/03/1797, Independent, Virginia
2. John Adams, 4/03/1797, 4/03/1801, Federalist, Massachusetts
3. Thomas Jefferson, 4/03/1801, 4/03/1809, Democratic-Republican, Virginia
   ...
42. Bill Clinton, 20/01/1993, 20/01/2001, Democratic, Arkansas
43. George W. Bush, 20/01/2001, 20/01/2009, Republican, Texas
List Operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;list&gt;.append(x)</code></td>
<td>Add element x to end of list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.sort()</code></td>
<td>Sort (order) the list. A comparison function may be passed as a parameter.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.reverse()</code></td>
<td>Reverse the list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.index(x)</code></td>
<td>Returns index of first occurrence of x.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.insert(i, x)</code></td>
<td>Insert x into list at index i.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.count(x)</code></td>
<td>Returns the number of occurrences of x in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.remove(x)</code></td>
<td>Deletes the first occurrence of x in list.</td>
</tr>
<tr>
<td><code>&lt;list&gt;.pop(i)</code></td>
<td>Deletes the ith element of the list and returns its value.</td>
</tr>
</tbody>
</table>

Taken from Zelle's slides: http://mcsp.wartburg.edu/zelle/python/PPICs2/slides/
Always start with *Small* Data Sets
Class Inheritance
Class Inheritance
The Idea...
Car class
- constructor
- draw
- move
- undraw
- etc...
Car class
- constructor
- draw
- move
- undraw
- etc...

Wanted with same features but different color
Car class
- constructor
- draw
- move
- undraw
- changeColor
- etc...

Wanted with same features but different color

Add new method
Car class
- constructor
- draw
- move
- undraw
- changeColor
- etc...

Wanted: new shape with same features
• We *could* write a completely new class for the new car shape…

• But instead, we can *save code, save time, save debugging aggravation*, by reusing the original car class.
• We **could** write a completely new class for the new car shape...

• But instead, we can **save code, save time, save debugging aggravation**, by reusing the original Car class.

• We will **derive** a new class from the Car class. The new class will be **derived** from it, and will **inherit** all the member variables and methods.

• The Car class will become the **super** class.
Another Way to Look at Inheritance
Another Way to Look at Inheritance

Super Class

Derived Class

Value

Value2
Coding Example
class Car:
    """A default simple car class. Sets the color to "Yellow" ""
    def __init__(self, rp):
        """builds a car 180 pixels long on a reference point""
        x = rp.getX()
y = rp.getY()
        length = 180        # geometry of the car
        height = length // 3
        firstQuarter = length/4
        thirdQuarter = length * 3 // 4
        p2 = Point(x+length, y+height)
selself.body = Rectangle(rp, p2)
selself.body.setFill("yellow")
selself.w1 = Wheel(Point(x+firstQuarter, y+height), 20,10)
selself.w2 = Wheel(Point(x+thirdQuarter, y+height), 20,10)

def draw(self, win):
    """draws the body and wheels on the window""
    self.body.draw(win)
selself.w1.draw(win)
selself.w2.draw(win)

def move(self, dx, dy):
    """moves the body and wheels some delta x and y""
    self.body.move(dx, dy)
selself.w1.move(dx, dy)
selself.w2.move(dx, dy)

def main():
    # open the window
    win = GraphWin("CSC111 Inheritance Demo", WIDTH, HEIGHT)

car = Car(Point(100,250))
car.draw(win)

win.getMouse()
win.close()
class Car:
    """A default simple car class. Sets the color to "yellow"""
    def __init__( self, rp ):
        """""""builds a car 180 pixels long on a reference point"""
        x = rp.getX()
y = rp.getY()
        length = 180
        # geometry of the car
        height = length // 3
        firstQuarter = length // 4
        thirdQuarter = length * 3 // 4
        p2 = Point( x+length, y+height )
self.body = Rectangle( rp, p2 )
self.body.setFill( "yellow" )
self.w1 = Wheel( Point( x+firstQuarter, y+height ), 20,10 )
sself.w2 = Wheel( Point( x+thirdQuarter, y+height ), 20,10 )

    def draw( self, win ):
        """draws the body and wheels on the window""
        self.body.draw( win )
sself.w1.draw( win )
sself.w2.draw( win )

    def move( self, dx, dy ):
        """moves the body and wheels some delta x and y""
        self.body.move( dx, dy )
sself.w1.move( dx, dy )
sself.w2.move( dx, dy )

class ColoredCar( Car ):
    """""""Class derived from Car. Allows resetting of body color"""
    def setColor( self, color ):
        """sets the color of the body""
        self.body.setFill( color )

def main():
    # open the window
    win = GraphWin( "CSC111 Inheritance Demo", WIDTH, HEIGHT )
car2 = ColoredCar( Point( 200, 100 ) )
car2.setColor( "lightgreen" )
car2.draw( win )
Creating a **Truck**
Derived from a **Car**
Two Different Syntaxes

```python
def draw(self, win):
    super().draw(win)
    self.top.draw(win)
```

```python
def draw(self, win):
    Car.draw(self, win)
    self.top.draw(win)
```
Two Different Syntaxes

```python
def draw(self, win):
    super().draw(win)
    self.top.draw(win)
```

```python
def draw(self, win):
    Car.draw(self, win)
    self.top.draw(win)
```

preferred

good when class inherits from two or more super classes
Graphics Libraries Are Built on Inheritance: Hierarchy of Classes
A Look at graphics.py

http://cs.smith.edu/dftwiki/index.php/Zelle%27s_Graphics.py_for_Python_3
Coding Exercise

• Create a new graphic class called MyCircle, that inherits from Circle, and that contains a circle and a label (text) inside the circle.

• MyCircle should support the following methods:
  • __init__(center, radius, text)
  • draw(win)
  • move(dx, dy)
  • setText(newText)
We stopped here last time...
Coding Exercise

• Create an **Animal** class (name, dob, vaccinated, tattooed)

• Create a sub-class of Animal: **Cow** (milkProduction)

• Create **Dog** a sub-class of Animal (trained, hospital, seeingDog, rescue)
Polymorphism
Polymorphism
Poly = Many
Morph = Form, Shape
Poly = Many
Morph = Form, Shape

Poly = Many
Morph = Form, Shape

Poly = Many
Morph = Form, Shape

Poly = Many
Morph = Form, Shape

Poly = Many
Morph = Form, Shape

Polymorphism is the ability for a function, or method, to behave \textbf{differently} depending on which class it belongs to, or what type of parameter is given to it.

Let's Code this!
class Animal:
    def name(self):
        return
    def sleep(self):
        print("sleeping now")
    def speak(self):
        return

class Dog(Animal):
    def name(self):
        print("I am a dog!")
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def name(self):
        print("I am a cat!")
    def speak(self):
        print("Meow!")

class Duck(Animal):
    def name(self):
        print("I am a duck!")
    def speak(self):
        print("Quack!")
```python
class Animal:
    def name(self):
        return
    def sleep(self):
        print("sleeping now")
    def speak(self):
        return

class Dog(Animal):
    def name(self):
        print("I am a dog!")
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def name(self):
        print("I am a cat!")
    def speak(self):
        print("Meow!")

class Duck(Animal):
    def name(self):
        print("I am a duck!")
    def speak(self):
        print("Quack!")

garfield = Cat()
rex = Dog()
donald = Duck()

garfield.name()
garfield.speak()
garfield.sleep()
rex.name()
rex.speak()
rex.sleep()
donald.name()
donald.speak()
```
```python
class Animal:
    def name(self):
        return
    def sleep(self):
        print("sleeping now")
    def speak(self):
        return

class Dog(Animal):
    def name(self):
        print("I am a dog!")
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def name(self):
        print("I am a cat!")
    def speak(self):
        print("Meow!")

garfield = Cat()
rex = Dog()
donald = Duck()

garfield.name()
garfield.speak()
garfield.sleep()

 rex.name()
 rex.speak()
 rex.sleep()

donald.name()
donald.speak()

>>> I am a cat!
Meow!
sleeping now
I am a dog!
Woof!
sleeping now
I am a duck!
Quack!
```
Polymorphism

```python
class Animal:
    def name(self):
        return
    def sleep(self):
        print( "sleeping now" )
    def speak(self):
        return

class Dog( Animal ):
    def name(self):
        print( "I am a dog!" )
    def speak(self):
        print( "Woof!" )

class Cat( Animal ):
    def name(self):
        print( "I am a cat!" )
    def speak(self):
        print( "Meow!" )

garfield = Cat()
rex = Dog()
donald = Duck()

garfield.name()
garfield.speak()
garfield.sleep()
rex.name()
rex.speak()
rex.sleep()
donald.name()
donald.speak()
```
Overloading

class Animal:
    def name(self):
        return
    def sleep(self):
        print( "sleeping now" )
    def speak(self):
        return

class Dog( Animal ):
    def name(self):
        print( "I am a dog!" )
    def speak(self):
        print( "Woof!" )

class Cat( Animal ):
    def name(self):
        print( "I am a cat!" )
    def speak(self):
        print( "Meow!" )

>>> garfield.name()
I am a cat!
>>> garfield.speak()
Meow!
>>> garfield.sleep()
sleeping now
>>> rex.name()
I am a dog!
>>> rex.speak()
Woof!
>>> rex.sleep()
sleeping now
>>> donald.name()
I am a duck!
>>> donald.speak()
Quack!
>>>
Another Example of Polymorphism: the + operator
Another Example of Polymorphism: the + operator

```python
>>> 1 + 10
11

>>> "Smith" + " " + "College"
'Smith College'

>>> ["Doc", "Grumpy", "Happy"] + ["Sleepy", "Bashful", "Sneezy", "Dopey"]
['Doc', 'Grumpy', 'Happy', 'Sleepy', 'Bashful', 'Sneezy', 'Dopey']

>>> ("Joe", 3) + ("Marie", 4)
('Joe', 3, 'Marie', 4)
```
Another Example of Polymorphism: the $+$ operator

```python
>>> 1 + 10
11

>>> "Smith" + " " + "College"
'Smith College'

>>> ["Doc", "Grumpy", "Happy"] + ["Sleepy", "Bashful", "Sneepy", "Dopey"]

>>> ("Joe", 3) + ("Marie", 4)
('Joe', 3, 'Marie', 4)
```
“Abundant syntax brings more burden than help.”

–Guido van Rossum
“Abundant syntax brings more burden than help.”

—Guido van Rossum