Creating Classes

Object Oriented Programming

Teaching Turtles New Tricks

```
class SmartTurtle(Turtle ):
    def drawSquare(self ):
        for i in range (0 ,4):
            self.turnRight ()
            self.forward ()
```

- The class Turtle exists.
- Here, we create a new kind of Turtle, a specialization called SmartTurtle, that knows how to draw squares.
- drawSquare is a method that SmartTurtle instances understand.
- All Python methods must accept self as the first parameter—self is the object receiving the message.

More Than One Method

```
class SmartTurtle(Turtle ):
  def drawSquare(self , width=100 ):
      for i in range (0 ,4):
          self.turnRight ()
          self.forward(width)
```

- Now SmartTurtle instances understand both how to drawSquare() and drawSquare(someWidth)

```
>>> mars = World()
>>> tina = SmartTurtle(mars)
>>> tina.drawSquare(30)
>>> tina.drawSquare(150)
>>> tina.drawSquare(100)
>>> # Does the same thing
>>> tina.drawSquare()
```

Inheritance and Overriding

- We can create a version of Turtle that's "confused."
  - Turns a random amount.
  - Goes forward a random amount.
- This class is a **subclass** of Turtle.
- That means it **inherits** everything from Turtle.
- It will **override** how to turn and go forward.

Confusedturtle

```
import random
class ConfusedTurtle(Turtle):
  def forward(self,num):
    Turtle.forward(self,int(num*random.random()))
  def turn(self,num):
    Turtle.turn(self,int(num*random.random()))


>>> pluto = World()
>>> goofy = ConfusedTurtle(pluto)
>>> goofy.forward(100)
>>> goofy.turn(90)
```

Slideshow

```
def playslideshow():
  pic = makePicture(getMediaPath("barbara.jpg"))
  snd = makeSound(getMediaPath("bassoon-c4.wav"))
  show(pic)
  blockingPlay(snd)
  pic = makePicture(getMediaPath("beach.jpg"))
  snd = makeSound(getMediaPath("bassoon-e4.wav"))
  show(pic)
  blockingPlay(snd)
  pic = makePicture(getMediaPath("santa.jpg"))
  snd = makeSound(getMediaPath("bassoon-g4.wav"))
  show(pic)
  blockingPlay(snd)
  pic = makePicture(getMediaPath("jungle2.jpg"))
  snd = makeSound(getMediaPath("bassoon-c4.wav"))
  show(pic)
  blockingPlay(snd)
```

- Functional (not OOP)
- From Procedural Abstraction:
  - We have duplicated code.
  - We should get rid of it.
- From Object-Oriented Programming:
  - We have an object: A slide.

Defining an Object (a Class)

- Objects know things.
  - Data that is internal to the object.
  - We often call those instance variables.
- Objects can do things.
  - Behavior that is internal to the object.
  - We call functions that are specific to an object methods.
    - But you knew that one already.
- We access both of these using dot notation
  - object.variable
  - object.method()

Classes

- Objects are instances of classes in many object-oriented languages.
  - Including Smalltalk, Java, JavaScript, and Python.
- A class defines the data and behavior of an object.
  - A class defines what all instances of that class know and can do.

E.g. The Slide Object

- What does a slide know?
  - It has a picture.
  - It has a sound
- What can a slide do?
  - Show itself.
    - Show its picture.
    - (Blocking) Play its sound.

Defining a show() Method

- To show a slide, we want to show() the picture and blockingPlay() the sound.
- We define the function as part of the class block.
  - So this is a def that gets indented.

Defining the Method show()

```
class slide:
  def show(self):
     show(self.picture)
     blockingPlay(self.sound)
```

- Why self?
  - When we say object.method(),
  - Python finds the method in the object's class, then calls it with the object as an input.
  - Python style is to call that self.
    - It's the object itself.
  - JavaScript uses "this" to call itself

Making It Simpler

- Can we get rid of those picture and sound assignments?
- What if we could call slide as if it were a real function, with inputs?
  - Then we could pass in the picture and sound filenames as inputs.
- We can do this, by defining what Java calls a constructor.
  - A method that builds your object for you.

Making Instances More Flexibly

- To create new instances with inputs, we must define a function named __init__
  - That's underscore-underscore-i-n-i-t-underscore-underscore.
  - It's the predefined name for a method that initializes new objects.
- Our __init__ function will take three inputs:
  - self, because all methods take that.
  - And a picture and sound filename.
    - We'll create the pictures and sounds in the method.

Now We Can Show Our Slide

```
>>> slide1.show()
```

- We execute the method using the same dot notation we've seen previously.
- Does just what you'd expect it to do.
  - Shows the picture.
  - Plays the sound.

Our Whole Slide Class

```
class slide:
  def __init__(self, pictureFile,soundFile):
      self.picture = makePicture(pictureFile)
      self.sound = makeSound(soundFile)

  def show(self):
      show(self.picture)
      blockingPlay(self.sound)
```

But Not Very “Object-Oriented”

- This version of the slide object is not very object oriented.
  - We set the picture and sound by directly touching the instance variables.
  - Adele Goldberg, who helped invent the first programming language Smalltalk, said that a key rule of object-oriented programming is, “Ask, don’t touch.”
- How do we let user’s “ask” to access instance variables?
  - By creating methods for getting (“getters”) and setting (“setters”) instance variables.

Class with Getters and Setters

```
class slide:
  def __init__(self, pictureFile,soundFile):
     self.setPicture(makePicture(pictureFile))
     self.setSound(makeSound(soundFile))

  def getPicture(self):
    return self.picture
  def getSound(self):
    return self.sound

  def setPicture(self,newPicture):
    self.picture = newPicture
  def setSound(self,newSound):
    self.sound = newSound

  def show(self):
     show(self.getPicture())
     blockingPlay(self.getSound())
```

Another Powerful Aspect of Objects: Polymorphism

- When the same method name can be applied to more than one object, we call that method polymorphic
  - From the Greek “many shaped”
- A polymorphic method is very powerful for the programmer.
  - You don’t need to know exactly what method is being executed.
  - You don’t even need to know exactly what object it is that you’re telling to show()
  - You just know your goal: Show this object!

More Methods Than Functions

- In general, there are many more methods defined in J E S than there are functions.
- Most specifically, there are a whole bunch of methods for drawing onto a picture that aren't defined as functions.
  - We simply ran out of time/energy to convert them all into functions.
  - And we rationalized that it was easier to deal with the complexity at the level of methods than functions.

Sunset Using Methods

- Any of our older functions will work just fine with methods.

```
def makeSunset(picture):
  for p in getPixels(picture):
    p.setBlue(p.getBlue()*0.7)
    p.setGreen(p.getGreen()*0.7)
```

Subclass Picture and Sound

- If we want to make our own methods on Picture and Sound, we have to subclass them.
- The new subclass will know everything from the Parent class.
  - But we can teach it new methods.

Making Sunset a Method

```
class MyPicture(Picture):
  def makeSunset(self):
    for p in getPixels(self):
      p.setBlue(int(p.getBlue()*0.7))
      p.setGreen(int (p.getGreen()*0.7))
```

```
>>> pict = MyPicture(getMediaPath("beach.jpg"))
>>> pict.explore()
>>> pict.makeSunset()
>>> pict.explore()
```

Making Reverse a Method

```
class MySound(Sound):
  def reverse(self):
    target = Sound(self.getLength())
    sourceIndex = self.getLength() - 1
    for targetIndex in range(0,target.getLength()):
      sourceValue = self.getSampleValueAt(sourceIndex)
      target.setSampleValueAt(targetIndex,sourceValue)
      sourceIndex = sourceIndex - 1
    return target
```

```
>>> sound = MySound(getMediaPath("always.wav"))
>>> sound.explore()
>>> target = sound.reverse()
>>> target.explore()
```

Why Objects?

- An important role for objects is to reduce the number of names that you have to remember.
  - writeSoundTo() and writePictureTo()
    vs.
    sound.writeTo() and picture.writeTo()
- They also make it easier to change data and behavior together.
  - Think about changing the name of an instance variable. What functions do you need to change? Odds are good that they're the ones right next to where you're changing the variable.
- Most significant power is in aggregation: Combining objects

Python Objects vs. Other Objects

- One of the key ideas for objects was “not messing with the innards.”
- Not true in Python.
  - We can always get at instance variables of objects.
- It is true in other object-oriented languages.
  - In Java or Smalltalk, instance variables are only accessible through methods (getPixel) or through special declarations (“This variable is public!”)

Inheritance

- We can declare one class to be inherited by another class.
- It provides instant polymorphism.
  - The child class immediately gets all the data and behavior of the parent class.
- The child can then add more than the parent class had.
  - This is called making the child a specialization of the parent.
  - A 3-D rectangle might know/do all that a rectangle does, plus some more:
- class rectangle3D(rectangle):

When Should You Use Objects?

- Define your own objects when you have:
  - Data in groups, like both pictures and sounds.
  - Behavior that you want to define over that group.
- Use existing objects:
  - Always—they're very powerful!
  - Unless you're not comfortable with dot notation and the idea of methods.
    - Then functions work just fine.