Introduction to Objected Oriented Design

So we've seen how to use Python to play with numbers, strings, and lists. There other sorts of goodies that come with Python, but it's somewhat limiting to have only those few data types at your disposal. One way to work around this is group together data with lists, and treat those like they were monolithic. This is a fast way to prototype basic functionality, but using lists gets tedious, as no one can tell what list thing is supposed to be what, and they tend to break functionality easily.

Python has a set of mechanisms that allow you to program in an object oriented way: in this way, Python enforces the way you treat objects (no abuse!) and tends to bring a bunch of other good things with it (like greater abstraction, which helps with refactoring. Don't worry too much about the benefits of object oriented programming, or OOP for short. They'll become apparent as you go).

Python is object oriented. This means that although we have been showing you can use python as a purely imperative language, you can also use it as an object oriented language (you can also use it as a functional language, but that's another story for another day...). If you have no idea what we just said, no problem, let's just go...

If you remember from last time we have all sorts of things to keep track of multiple things (lists, tuples). However, trying to keep things together in lists isn't the best way to do it: lists then are brittle and prone to break, and you have to remember which index is which thing [however, it is pretty good for rapid prototyping. But at some point in time]. it's easier to wrap up everything into a monolithic object that you can toss around and ask to do things for you. hence, object oriented programming.

Classes

We'll start out with the very barest of bones class definition:

```
class Banana:
      pass
```

this tells python that you're defining a new type of object, a Banana. You can make Bananas, and Python will know it's a Banana

```
b = Banana()
type(b)
```

tells us that b is a Banana, but it's indistinguishable from any other empty object that we might create, like an empty Apple class.

the problem is that we haven't told Python what separates the Banana class from any other class that we might make. We need to give our Banana some attributes, like, say, whether or not it is peeled:

```
class Banana:
      def __init__(self, peeled):
            self.peeled=peeled
```

As you should remember, the

```
def __init__():
```

makes a new function. Since it's inside the class definition, we call it a method. This particular method is marked as special, due to the double _ surrounding the name, as it is used by Python to set up the Banana objects when you make them. in this case, we give the object a peeled attribute: a variable named peeled that is attached to every Banana object.

Note: the self is magical voodoo stuff, don't worry about it to much. we'll tell you about it later

we'll make a couple of bananas:

```
joes_banana = Banana(True)
anns_banana = Banana(False)
joes_banana.peeled
anns_banana.peeled
```

what happens when we don't initialize with a parameter?

```
martins_banana = Banana()
```

TIS AN ERROR. the __init__ method expects a peeled parameter, and doesn't get one, much like a normal function

a useful thing is to include default values for your parameters (Note that this applies to all functions, not just methods)

```
class Banana:
      def __init__(self, peeled=False):
            self.peeled=peeled
```

now we can do

```
bobs_banana = Banana()
bobs_banana.peeled
```

now, let’s add more parameters and more instance variables, to make our Banana class more detailed

```
class Banana:
      def __init__(self,peeled=False,eaten=False):
            self.peeled=peeled
            self.eaten=eaten
```

now, note how the keyword parameters work. since they already have default values, we can be quite flexible about which parameters we pass

```
zoe_banana = Banana(peeled=True)
zoe_banana.peeled,mya_banana.eaten
mya_banana = Banana(eaten=True,peeled=True)
mya_banana.peeled,mya_banana.eaten
sven_banana = Banana(True,True)
sven_banana.peeled, sven_banana.eaten
```

Attributes are well and good, but an object has things done to it, too. We can add more methods than just __init__ that also operate on Banana objects.

```
class Banana:
      def __init__(self,peeled=False,eaten=False):
            self.peeled=peeled
            self.eaten=eaten

      def peel(self):
            if(not(self.peeled)):
                  self.peeled=True
            else:
                  print(“This banana is already peeled!”)

charlie_banana = Banana()
charlie_banana.peeled
charlie_banana.peel()
charlie_banana.peeled
```

explain self while you explain the example

easy way to think about it: self gets replaced by the instance’s name
my_instance = My_Class()

my_instance.function(parameter) is equivalent to
My_Class.function(my_instance, parameter)

here’s a full Banana class def:

```
class Banana:
      def __init__(self,peeled=False,eaten=False):
            self.peeled=peeled
            self.eaten=eaten
            self.name = “”

      def peel(self):
            if(not(self.peeled)):
                  self.peeled=True
            else:
                  print(“This banana is already peeled!”)

      def eat(self):
            if(self.peeled and not(self.eaten)):
                  self.eaten=True
                  return True
            else:
                  return False
      def suck_through_straw(self):
            self.eaten = True
      def give_name(self,name):
            self.name = name
      def ask_name(self):
            if(self.name):
                  print(“My name is ”+name+”!”)
            else:
                  if(not(self.eaten)):
                        print(“I am unloved and uneaten”)
                  else:
                        print(“I am unloved, but eaten”)

class BananaEater:
      def __init__(self):
            self.sick=False

      def eat(self,banana):
            if(banana.eat()):
                  if (banana.isRipe()):
                        print(“Wow that was delicious”)
                  else:
                        print(“Oh god, I’m so sick”)
                        self.sick = True
            else:
                  print(“This banana is already eaten”)
```

IF NECESSARY, we can end here

Work through example(s) with audience interaction. If the audience can’t come up with anything:

class Pirate:

    have them fight with each other (type checking)

class Parrot:

    screams head off, says “Polly wants a cracker”, or “Help! They’ve turned me into a parrot”