Inheritance in Python

In the last few lectures we were introduced to classes in Python and Pygame, as well as how to create a program that uses multiple files. In the examples shown, we saw two classes that seemed fairly similar:

1. token
2. pictoken

In fact, as the names suggest, the latter IS-A type of the former. If we look at the code for the latter, it's awful similar to the former with just a couple pieces added. Just as we write regular functions so we can reuse code and avoid rewriting it, this seems to be another opportunity to write code in a different way so that maybe we can reuse some of it. Fundamentally, the idea is this:

Put all the functionality of any token in the token class. Then, when writing the pictoken class, indicate that this class is inheriting from token (so it gets to reuse all of its code), but if there are some methods that should work differently (for example draw, where we want to draw the image and not a rectangle), then rewrite those methods, or if there should be new methods, write those.

Let's first illustrate the syntax and power of inheritance through a traditional Python example (without Pygame). The classical example given in every textbook of inheritance is an employee class. For simplicity, even though we know how to write a program in multiple files, this example will be all in one file. Our basic employee will have the following instance variables:

1. First Name
2. Last Name
3. ID number
4. Hourly pay rate
5. Total pay

We'll just include two methods beyond a constructor:

1. __str__ (for a string representation of the object)
2. pay (this represents paying the employee for some work)
3. bonus (this is for a one time bonus)

Consider a special employee of class commissionemployee who, in addition to hourly pay, receives a fixed amount for each item she sells. This class will have two additional instance variables:

1. Number of items sold
2. Bonus per item sold

Here is the whole employee class:

```
class employee:

    firstName = ""
    lastName = ""
    id = 0
    payPerHour = 10.00
    totalPay = 0

    # Standard constructor.
    def __init__(self,fName,lName,myid,payPH):
        self.firstName = fName
        self.lastName = lName
        self.id = myid
        self.payPerHour = payPH
        self.totalPay = 0

    # We'll just print name and total pay here.
    def __str__(self):
        return self.lastName+", "+self.firstName+" Total Pay =
"+str(self.totalPay)

    # To keep it simple, no overtime!
    def pay(self,hours):
        self.totalPay += hours*self.payPerHour

    # This is for a straight one time bonus of add dollars.
    def bonus(self,add):
        self.totalPay += add
```

Here is a basic main method to test this class:

```
def main():

    aleena = employee("Aleena", "Patel", 2233447, 12)
    geno = employee("Geno", "Lin", 8675432, 11)
    workers = []
    workers.append(aleena)
    workers.append(geno)

    workers[0].pay(20)
    workers[1].pay(10)
    for w in workers:
        print(w)

    for w in workers:
        w.bonus(50)
    for w in workers:
        print(w)
```

Here is the corresponding output that shows the first payment and then what the pay looks like after the bonus:

```
Patel, Aleena Total Pay = 240
Lin, Geno Total Pay = 110
Patel, Aleena Total Pay = 290
Lin, Geno Total Pay = 160
```

For the commissionemployee class, we won't define the bonus method at all because we want to inherit the bonus method from the employee class. For the constructor, the __str__ method and the pay method, we will write our own method, but we will use the special keyword super to call the corresponding methods from the base class (the class from which we inherit) to shorten our code.

First, here is how we indicate that one class (commissionemployee) is inheriting from another (employee):

```
class commissionemployee(employee):
```

It's common to call employee the base class and commissionemployee the subclass. We also say that a commisionemployee IS-A employee.

Syntax-wise, if we want to call a method from the class we are inheriting from we say:

```
super().methodname(parameters)
```

This way, we don't have to rewrite that code, since it's the same. Here is the whole commissionemployee class:

```
class commissionemployee(employee):

    numSold = 0
    bonusPerItem = 0

    def __init__(self,fName,lName,myid,payPH,payPI):
        super().__init__(fName,lName,myid,payPH)
        self.numSold = 0
        self.bonusPerItem = payPI

    def __str__(self):
        return super().__str__()+" Items Sold = "+str(self.numSold)

    def pay(self,hours,newitems):
        super().pay(hours)
        self.numSold += newitems
        self.totalPay += newitems*self.bonusPerItem
```

Let’s test this out. Here is the rest of main after the previous initial test shown:

```
    # Now let's test one commission employee.
    print("Just test our new commisionemployee Jahmal.")
    workers.append(commissionemployee("Jahmal","Moss",3434345,20,80))
    workers[2].pay(10,3)
    print(workers[2])
    print()

    # Add another bonus to all.
    print("One final bonus to all.")
    for w in workers:
        w.bonus(100)
    for w in workers:
        print(w)
```

Logically, what should happen here is that Jahmal gets paid 200 dollars for his hourly work and 240 dollars for his sales. Indeed this is the case as the corresponding output is:

```
Just test our new commisionemployee Jahmal.
Moss, Jahmal Total Pay = 440 Items Sold = 3
```

Finally, all three receive the same bonus. Also, notice that we get to call the SAME EXACT __str__ method on all three employees when we print, but, Python is smart enough to dynamically (at run time), figure out that the first two employees are just regular ones (so it calls __str__ in the employee class), but that the last employee is really a commissionemployee. This is called polymorphism, where you can have a list of all employees, but at run time, Python will detect if any of these are really objects from the subclass and it will call the appropriate method accordingly.

Applying Inheritance to Pygame (Fruit Game Example)

Now, let's apply the idea of inheritance to the fruit game. Let's review the token class quickly, with imports omitted:

```
class token:

    dx = 0
    dy = 0
    rec = None
    color = pygame.Color(0,0,255)

    def __init__(self,myx,myy,mydx,mydy,width,height,mycolor):
        self.dx = mydx
        self.dy = mydy
        self.rec = pygame.Rect(myx, myy, width, height)
        self.color = mycolor

    def move(self):
        self.rec.x += self.dx
        self.rec.y += self.dy

    def bounceLeft(self):
        if self.rec.x + self.dx < 0:
            self.dx = -self.dx

    def bounceRight(self, SCREEN_W):
        if self.rec.x + self.dx > SCREEN_W-self.rec.width:
            self.dx = -self.dx

    def bounceUp(self):
        if self.rec.y + self.dy < 0:
            self.dy = -self.dy

    def bounceDown(self, SCREEN_H):
        if self.rec.y + self.dy > SCREEN_H-self.rec.width:
            self.dy = -self.dy

    def updateFrame(self, DISPLAYSURF):
        self.move()
        self.bounceLeft()
        self.bounceRight(DISPLAYSURF.get_width())
        self.bounceUp()
        self.bounceDown(DISPLAYSURF.get_height())
        self.draw(DISPLAYSURF)

    def draw(self, DISPLAYSURF):
        pygame.draw.rect(DISPLAYSURF, self.color, self.rec, 0)
```

For our purposes, we'll reuse the constructor, move and draw only, because we don't want the fruit bouncing off of the walls. We don't even have to define move, because this will be the exact same. We'll override the other two, using super only for the constructor and not draw. Finally, we'll add a hit method that will detect if a single position is inside the rectangle defined by the picitem. Here is the whole picitem class:

```
class pictoken(token):

    pic = None
    pts = 0

    def __init__(self,myx,myy,mydx,mydy,mycolor,mypic,mypts):

super().__init__(myx,myy,mydx,mydy,mypic.get_width(),mypic.get_height(),mycol
or)
        self.pic = mypic
        self.pts = mypts

    def hit(self, mypos):
        return self.rec.collidepoint(mypos)

    def draw(self, DISPLAYSURF):
        DISPLAYSURF.blit(self.pic,(self.rec.x, self.rec.y))
```

With this class written this way, we'll just make our objects in fruitgame of time pictoken and everything else will be the same, so there will be no changes to fruitgame.py compared to the previous example and it'll work the same way as before!