BME646/ ECE695DL:Homework 1

Aditya Chauhan

17 January 2021

1 Introduction

In this homework Basics of Python Object Oriented Programming are covered by using a Parent and Child Class. We go over a Parent Class *Countries* with two instance variables and one function and than define a child class *GeoCountry* with inheritance of two variables from Parent class and two of it's own variables. Plus in child class we also expand it have three functions and overwrite one of the Parent class function.

2 Methodology

We defined two classes *Countries* and *GeoCountry*. Class Countries holds the information of the birth, death and last year count of population in a **List[int]** population(of size 3) variable and **string** variable capital to hold name. Countries class also expands to a function net population that uses the Countries instance variable to calculate the current net population (birth - death + last count) using equation below.

$$currentNet = self.population[0] - self.population[1] + self.population[2] \tag{1}$$

Second class GeoCountry is made as a child class of Countries and inherits instance variable of parent class. It also has two of it's own instance variables density and area. area is passed during object initialization of class GeoCountry and density is set to zero at this point by default. There is also a scenario where we overwrite a function from parent class into child class and there we expand the length of inherited population variable to 4 from 3. Other functions in GeoCountry are two version of calculating density and one function that take the choice from user between which method to chose for density calculation. Also by the end we updates the current net population(birth - death + (second last count + last count)/2) calculation method to work with 4 length population List.

$$currentNet = self.population[0] - self.population[1] + (self.population[2] + self.population[3])/2 \tag{2}$$

Finally the function that takes the choice of user to calculate the density return the function instance and that instance can be used by adding **() operator** also called a function call operator.

3 Implementation and Results

3.1 Class Countries

```
class Countries:
    def __init__(self, capital, population):
        self.capital = capital
        self.population = population

    def net_population(self):
        current_net = self.population[0]-self.population[1]+self.population[2]
        return current_net
```

Figure 1: Class Countries Python Implementation

In this class(see Figure 1) we have a method to initialize the object of this class with two instance variables capital and population which are given as a input during the initialization.

Next we have a function that takes the population and calculates the net population of this year (check equation 1).

Below are the input and result

1. ob = Countries('AAA', [20,100,1000])
2. print(ob.netPopulation())
3. Result = 920

3.2 Class GeoCountry

3.2.1 init function

```
def __init__(self,capital, population, area):
    super().__init__(capital, population)
    self.area = area
    self.density = 0
```

Figure 2: GeoCountry Init function

Here we defined the init function(see Figure 2) that inherits the instance variable from Class Countries. *super()* is used to accomplish this task and the use of this can be seen in other function calls of this Class GeoCountry

3.2.2 density calculator 1

```
def density_calculator1(self):
    if len(self.population)==3:
        self.density = super().net_population()/self.area
    elif len(self.population)==4:
        self.density = (self.population[0] - self.population[1] +
                        (self.population[2] + self.population[3]) / 2)/self.area
    return self.density
```

Figure 3: Density Calculator function 1

In this implementation(see Figure 3) we take to a basic density calculation where the data provided didn't have any form of error associated with it. We take two cases of length of population is 3 or 4. If 3 we used the current net population calculation mentioned in equation 1. If we have population length 4 we use equation 2 for current net population calculation.

3.2.3 density calculator 2

```
def density_calculator2(self):
    if len(self.population)==3:
        self.population[2] = self.population[2] - self.population[0] + self.population[1]
        self.density = super().net_population() / self.area
    elif len(self.population)==4:
        self.population[3] = 2*(self.population[3] - self.population[0] + self.population[1])-self.population[2]
        self.density = (self.population[0] - self.population[1] + (
                    self.population[2] + self.population[3]) / 2) / self.area
    return self.density
```

Figure 4: Density Calculator function 2

In this implementation(see Figure 4) we have an error in data as the last population was updated by current net population and we rectify it using equation 3 or equation 4 based on the length of population List. We take two cases of length of population is 3 or 4. If 3 we used the current net population calculation mentioned in equation 1. If we have population length 4 we use equation 2 for current net population calculation.

$$self.population[2] = self.population[2] - self.population[0] + self.population[1] \tag{3}$$

$$self.population[3] = 2*(self.population[3] - self.population[0] + self.population[1]) - self.population[2] \tag{4}$$

3.2.4 net density

```
def net_density(self,choice):
    out = self.density_calculator1 if choice==1 else self.density_calculator2
    return out
```

Figure 5: Net Density function

In this function(see Figure 5) we take a choice argument between 1 or 2. If it's 1 we calculate density using density calculator 1. If it's 1 we calculate density using density calculator 2. In both these cases we return the function instance and than we add **() function operator** to these function call objects and get result.

3.2.5 net population

```
def net_population(self):
    current_net=0
    if len(self.population)==3:
        self.population.append(self.population[0] - self.population[1] + self.population[2])
        current_net = self.population[0] - self.population[1] + (self.population[2] + self.population[3]) / 2
    elif len(self.population)==4:
        current_net = self.population[0] - self.population[1] + (self.population[2]+self.population[3])/2
        self.population[2] = self.population[3]
        self.population[3] = current_net
    return current_net
```

Figure 6: Overridden Net Population function

Through this function(see Figure 6) we override the net population function in Class Countries. Here we increase the size of population list from 3 to 4 by appending the current population after the last population and in each iteration we keep doing this by replacing the second last net with last net and append current net after that, thus keeping size to 4.

Refer below(see Figure (7b)) to see how this works in multiple net population calls.

```
ob1 = GeoCountry('AA', [55,10,25,70],5)
print(ob1.net_population())
print(ob1.net_population())
print(ob1.net_population())
```

(a) Net Population update test case

```
92.5
126.25
154.375
```

(b) Result of multiple net population calls

3.2.6 Results

```
if __name__ == "__main__":
    ob = Countries('AAA', [20,100,1000])
    print("Net population for Countries object", ob.net_population())
    ob1 = GeoCountry('YYY', [20, 100, 1000], 5)
    print("Density after initialization of geoCountry class: ",ob1.density)  # 0
    print("Population after initialization of geoCountry class: ",ob1.population)  # [20,100,1000]
    ob1.density_calculator1()
    # Here the net_population of class Countries is used
    print("Density based on density calculator 1 for ob1: ", ob1.density)  # 184.0
    ob1.density_calculator2()
    # Here the net_population of class Countries is used
    print("Population after density calculator 2 for ob1: ",ob1.population)  # [20, 100, 1080]
    print("Density after density calculator 2 for ob1: ",ob1.density)  # 200.0
    ob2 = GeoCountry('ZZZ', [20, 50, 100], 12)
    fun = ob2.net_density(2)
    print("Density after density calculator 2 for ob2: ",ob2.density)  # 0
    fun()
    print("{:.2f}".format(ob2.density))  # 8.33
    print("Current population after above calls: ",ob1.population)  # [20,100, 1080]
    print("Calling net_population overridden version",ob1.net_population())  # 960.0
    print("Population list size increases: ",ob1.population)  # [20,100,1080,1000]
    print("Current density till above calls itself: ", ob1.density)  # 200.0
    ob1.density_calculator1()
    # Here the overridden net_population is used
    print("Population after density calculator 1 for ob1 and pop len 4: ", ob1.population)  # [20, 100, 1080, 1000]
    print("Density after density calculator 1 for ob1 and pop len 4: ",ob1.density)  # 192.0
    ob1.density_calculator2()
    # Here the overridden net_population is used
    print("Population after density calculator 2 for ob1 and pop len 4: ",ob1.population)  # [20, 100, 1080, 1080]
    print("Population after density calculator 2 for ob1 and pop len 4: ",ob1.density)  # 200
```

Figure 8: test cases to test different cases

```
Net population for Countries object 920
Density after initialization of geoCountry class:  0
Population after initialization of geoCountry class:  [20, 100, 1000]
Density based on density calculator 1 for ob1:  184.0
Population after density calculator 2 for ob1:  [20, 100, 1080]
Density after density calculator 2 for ob1:  200.0
Density after density calculator 2 for ob2:  0
Density of ob2: 8.33
Current population after above calls:  [20, 100, 1080]
Calling net_population overridden version 960.0
Population list size increases:  [20, 100, 1080, 1000]
Current density till above calls itself:  200.0
Population after density calculator 1 for ob1 and pop len 4:  [20, 100, 1080, 1000]
Density after density calculator 1 for ob1 and pop len 4:  192.0
Population after density calculator 2 for ob1 and pop len 4:  [20, 100, 1080, 1080]
Population after density calculator 2 for ob1 and pop len 4:  200.0
```

Figure 9: Results of the test case

4 Lessons Learned

Covered the basics of how OOP programming works with scripting language and how robust coding can be done with these concepts.