Exercise: the zoo

Write a Python program that simulates a zoo. In particular, create different modules in different files as follows.

- Class **Animal**:
  - Attributes:
    - string: name
    - int: age
  - Methods:
    - info (does nothing)
    - speaks (does nothing)
    - moves (does nothing)
    - eats (does nothing)
    - drinks (does nothing)
    - sleeps (takes an integer and waits for n seconds)
    - getter/setter for name name and age

- Class **Dog**: subclass of **Animal**
  - Attributes:
    - string: breed
  - Methods:
    - info (returns the breed)
    - speaks (returns the string “barks”)
    - moves (returns the string “runs”)
    - eats (returns the string “eats”)
    - drinks (returns the string “drinks”)
    - getter/setter for the breed

- Class **Horse**: subclass of **Animal**
  - Attributes:
    - string: color
  - Methods:
    - info (returns the color)
    - speaks (returns the string “neigh”)
    - moves (returns the string “gallops”)
    - eats (returns the string “eats”)
    - drinks (returns the string “water”)
    - getter/setter for color

- Classe **Lion**: subclass of **Animal**
    - Attributes:
        - int: weight
    - Methods
        - info (returns the weight)
        - speaks (che ritorna la stringa “roar”)
        - moves (che ritorna la stringa “runs fast”)
        - eats (che ritorna la stringa “devour”)
        - drinks (che ritorna la stringa “gobble”)
        - getter/setter for weight

The main program generated a random integre between 1 and 10 objects of class **Animal**, using randomly chosen names and numbers.

Next, implement a cycle of 20 iterations where you choose randomly one of the 20 object and a random operation to apply over the chosen object.

In each iteration, print a string reporting the name of the animal, its age, the information of the animal, and the chosen operation. After printing the string, the program pauses for 1s before starting the next iteration.

Names can be stored into an array; to generate random numbers you can use functions **random**; for pauses, refer to the function **sleep** within module **time**.

Exercise: Point3D

Write a class **Point3D** that represents a point in a 3D Euclidean apace. In particular, that class contains:

1. the constructor (__**init**__) with three parameters (x, y, z) that have 0 as default values;
2. the method **distance**(self, *point*) that returns the distance of to *point*;
3. the definition of the special method __**repr**__(self) that returns a string that represents the point; for instance, "Point3D(x, y, z)" where x, y, and z are the coordinates of the point;
4. the definition of the methods __**eq**__(self, *point*), __**lt**__(self, *point*), and __**gt**__(self, *point*), that returns a boolean stating whether the two points are equal, or this point is smaller or greater than *point*. A point p1 is greater than a point p2 if the distance of p1 from the origin is greater than the distance of p2 from the origin.

Write a main where you create points and use all these methods.
NOTE: recall that methods __**eq**__, __**lt**__ and __**gt**__ are automatically called when you use the operators ==, < and >, respectively. __repr__ is called when you *print* on object.

Exercise: Sphere3D

Write a class **Sphere3D** that represents a sphere in a 3D Euclidean space. In particular, that class contains:

1. the constructor (**__init__**) with two parameters (*center*, *radius*), where *center* is a **Point3D** that represents the center, with default value the origin, and *radius* is the values of the radius, with default value 1. All'interno del costruttore calcolare anche superficie e volume della sfera;

2. the definition of method **__repr__**(self), similar to the corresponding method of class **Point3D**;

3. the definition of methods **__eq__**(self, *sphere*), **__lt__**(self, *sphere*) and **__gt__**(self, *sphere*), that given a *sphere* returns a boolean that represents whether the two spheres are equal, or if this sphere has a volume smaller or greater than *sphere*;

4. the definition of methods **contains**(self, *point*) that returns **TRUE** whether *point* is contained in the sphere;

5. the definition of method **intersect**(self, *sphere*) that returns TRUE whether this sphere is intersected by *sphere*.

Write a main that uses all of the above methods.

Exercise: combine *Point* and *Sphere*

Finally, define a main that uses **Point3D** and **Sphere3D**, that randomly generates 20 spheres, having as radius a real number between [1, 3] and with center having integer coordinates x, y, z ∈ [0, 10], and 40 points having integer coordinates x, y, z ∈ [0, 10]. After generating all spheres and points, look for:

1. the sphere that contains most of the points; if there is more than one sphere, returns the smallest one; if again there is more than one sphere satisfing this conditions, returns the closest to the origin;
2. the sphere that intersects most of the spheres; if there is more than one, follow rules as in the previous case;
3. the point that is contained in most of the spheres; is there is more than one, returns the largest point, according to the definition of the methiod **__gt__**.