



DATA STRUCTURES ASSIGNMENT: PET STORE

walkthrough
emily nie



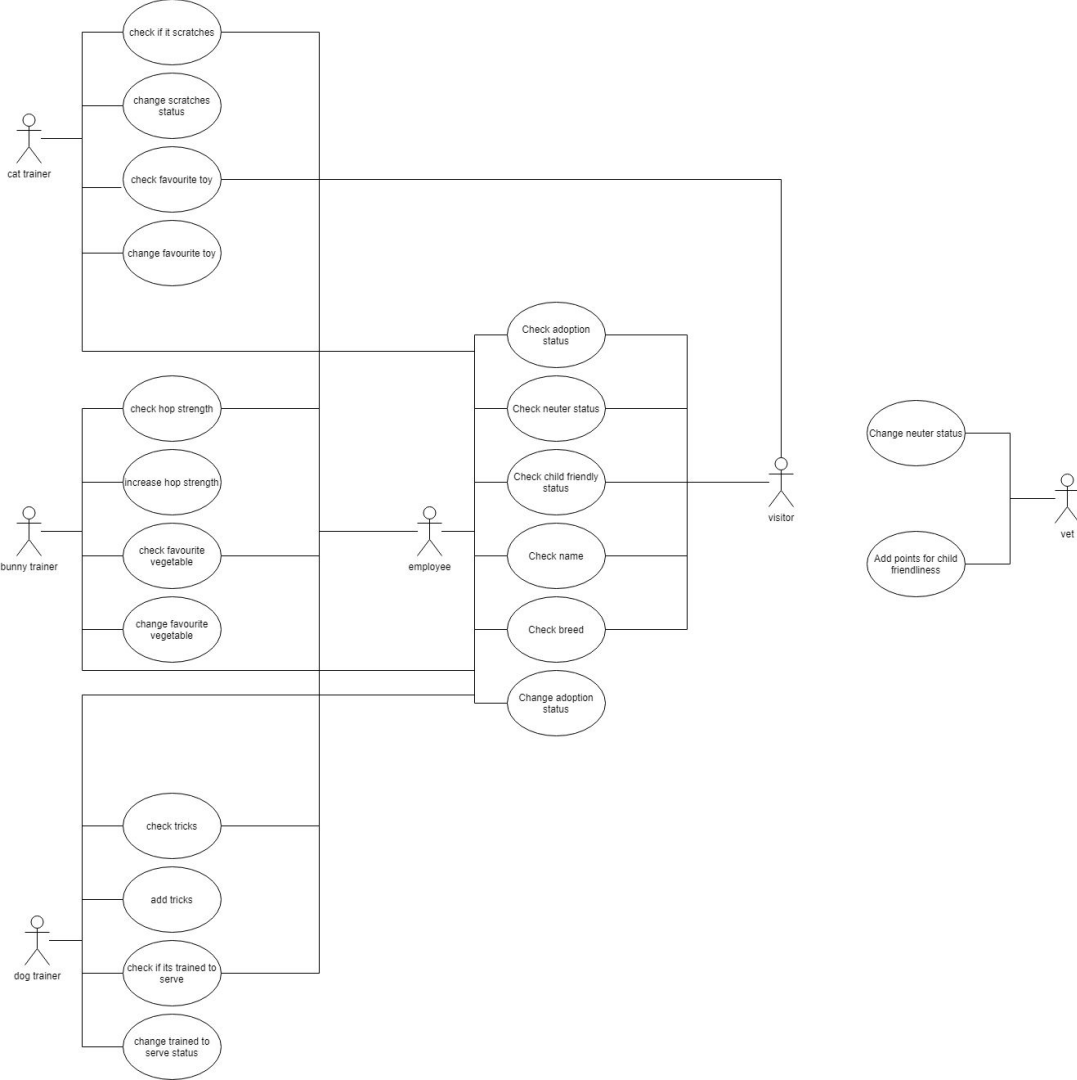
UML DIAGRAMS

Before starting any of the coding, I went ahead and created my UML diagrams. I used them to brainstorm different ideas, and see which ones I could expand and which ones wouldn't work very well

I ended up coming up with the theme of animals, and from there I used the UML Class diagram to come up with classes whose attributes/methods interconnected. This allowed me to easily determine the types of variables I needed when coding, as well as set up inheritance structures. Overall, this was essential to complete before coding, as it gave me a clear direction to follow.

The highlighted portions are the portions i ended up coding (I left out the Wild part as it would be hard to incorporate wild and pet animals into a single scenario). As for the get methods, those were combined into a `__repr__` method that displayed the attributes in an organized format.

Note: though the diagram has private and public classes, Python does not have them and all attributes in the code are public



UML DIAGRAMS

The UML Use Case diagram allowed me to narrow down on which attributes and qualities made sense to implement, based on which ones would have a purpose in real life

When coding the main.py portion of my program, this diagram helped me think of a realistic scenario where I could instantiate objects and call on their methods.

OBJECTS

01

Classes

Blueprint for a group of individuals

When it comes to the actual coding, the program is object oriented.

The two big ideas that are involved in this program would be the relationship between the classes and objects. The classes act as blueprints, defining the attributes of a group and the actions (methods) that the group can do. When you take that blueprint and use it to create a unique individual that still follows those specific guidelines, you create an instance of the class or an object.

02

Object

Unique individual that follows that blueprint

Classes and objects allow for reusability and as a result less repetition when coding.

USING OBJECTS IN PROGRAM + WHY?

Bunnies

```
1 Elsa;False;white;red;9;Polish;False;True;10;8;corn
2 Anna;False;yellow;brown;6;Mini Rex;True;True;8;3;broccoli
3 Olaf;True;white;black;2;Rex;True;True;9;1;carrot
```

Class

```
class Bunny(Pet):
```

Constructor

```
def __init__(self, name, gender, furColor, eyeColor, age, breed, forAdoption = True,
             neutered = False, childFriendly = 5, hopStrength = 1, favouriteVegetable =
             "carrot"):
```

Creating multiple objects

```
bunnies.append(Bunny(name, gender, furColor, eyeColor, age, breed, forAdoption,
                     neutered, childFriendly, hopStrength, favouriteVegetable))
```

In my program, there was a lot of repeating elements, especially with the pets. In just the bunny section, you had 3 different individuals that needed to be created.

All these bunnies share common traits like a name, gender, fur color and eye color. Individual bunnies will have different names, genders, fur and eye colors, but they all have these same attributes. They also share common activities such as eating and hopping.

Therefore to get rid of redundancy, classes and objects can be used. As long as the necessary parameters are supplied, objects can be created with a constructor and a line that uses that constructor. Objects then can use all the methods in the class.

USING OBJECTS IN PROGRAM + WHY?

Bunnies

```
1 Elsa;False;white;red;9;Polish;False;True;10;8;corn
2 Anna;False;yellow;brown;6;Mini Rex;True;True;8;3;broccoli
3 Olaf;True;white;black;2;Rex;True;True;9;1;carrot
```

Class

```
class Bunny(Pet):
```

Constructor

```
def __init__(self, name, gender, furColor, eyeColor, age, breed, forAdoption = True,
             neutered = False, childFriendly = 5, hopStrength = 1, favouriteVegetable =
             "carrot"):
```

Creating multiple objects

```
bunnies.append(Bunny(name, gender, furColor, eyeColor, age, breed, forAdoption,
                     neutered, childFriendly, hopStrength, favouriteVegetable))
```

This is much more efficient than creating tons of separate variables for each bunny. If we wanted to make the bunny do something (such as hopping), we can call the method instead of repeating the same executable lines for each bunny.

Objects also make the program cleaner and easier to follow for the one coding. If more pets arrive, we can easily account for them by creating another instance of the related class.

In this particular program, an array is used to hold the different objects, allowing for easy access.

EXTENDING OBJECTS

Bunny

```
def __init__(self, name, gender, furColor, eyeColor, age, breed, forAdoption = True,
             neutered = False, childFriendly = 5, hopStrength = 1, favouriteVegetable =
             "carrot"):
```

Dog

```
def __init__(self, name, gender, furColor, eyeColor, age, breed, forAdoption = True,
             neutered = False, childFriendly = 5, trainedToServe = False, tricks = []):
```

Cat

```
def __init__(self, name, gender, furColor, eyeColor, age, breed, forAdoption = True,
             neutered = False, childFriendly = 5, scratches = False, favouriteToy = "yarn"):
```

All pet bunnies have a name, age, color, etc., but so do dogs and cats. All 3 types of pets also all can do things such as eat and grow one year older.

This is where inheritance comes to play. Instead of listing out these same attributes and methods for each class, we can have an overarching parent class from which children class inherit attributes and methods. Bunnies, dogs and cats all share attributes and methods, so they can all inherit from an overarching class called Pet. The Pet class then inherits attributes and methods from a general Animal class.

EXTENDING OBJECTS - WHY IT WAS NEEDED

Animal

```
self.gender = gender
self.furColor = furColor
self.eyeColor = eyeColor
self.age = age
```

Pet

```
super().__init__(gender, furColor, eyeColor, age)
self.name = name
self.breed = breed
self.forAdoption = forAdoption
self.neutered = neutered
self.childFriendly = int(childFriendly)
```

Cat

```
super().__init__(name, gender, furColor, eyeColor, age, breed, forAdoption,
neutered, childFriendly)
self.trainedToServe = trainedToServe
self.tricks = tricks
```

Dog

```
self.scratches = scratches
self.favouriteToy = favouriteToy
super().__init__(name, gender, furColor, eyeColor, age, breed, forAdoption,
neutered, childFriendly)
```

Bunny

```
self.hopStrength = int(hopStrength)
self.favouriteVegetable = favouriteVegetable
super().__init__(name, gender, furColor, eyeColor, age, breed, forAdoption,
neutered, childFriendly)
```

Since there were so many types of animals, it would have been inefficient to write out the same list of attributes/methods in different classes, and extremely repetitive if the program was developed further.

By using inheritance, all methods of the parent class can be accessed without creating them again (unless you want to override it). As pictured, this also saves a lot of space when creating constructors, as inherited attributes don't need to be listed again.

This also ensures that all children will have a uniform structure, and makes the program a lot more modifiable. If we were to add more pet animal classes like birds or hamsters, we could do so very quickly by inheriting values from the Pet class.

*Though there is no second child class under the animal class, it still makes things neater and allows for the addition of another class if needed (ex. If i were to implement the Wild class from the UML, it could have been done very quickly)


FILE READING/WRITING AND HOW WAS IT EASIER

When creating the objects, data was read from .txt files, which saves a lot of time. If I need to add in a pet, all I have to do is type in its attributes to the .txt file and the for loop will use it to create an object. This is much easier compared to going in the code and manually creating an object. This is also much more realistic as in real life you are not going to manually type in parameters to a code file.

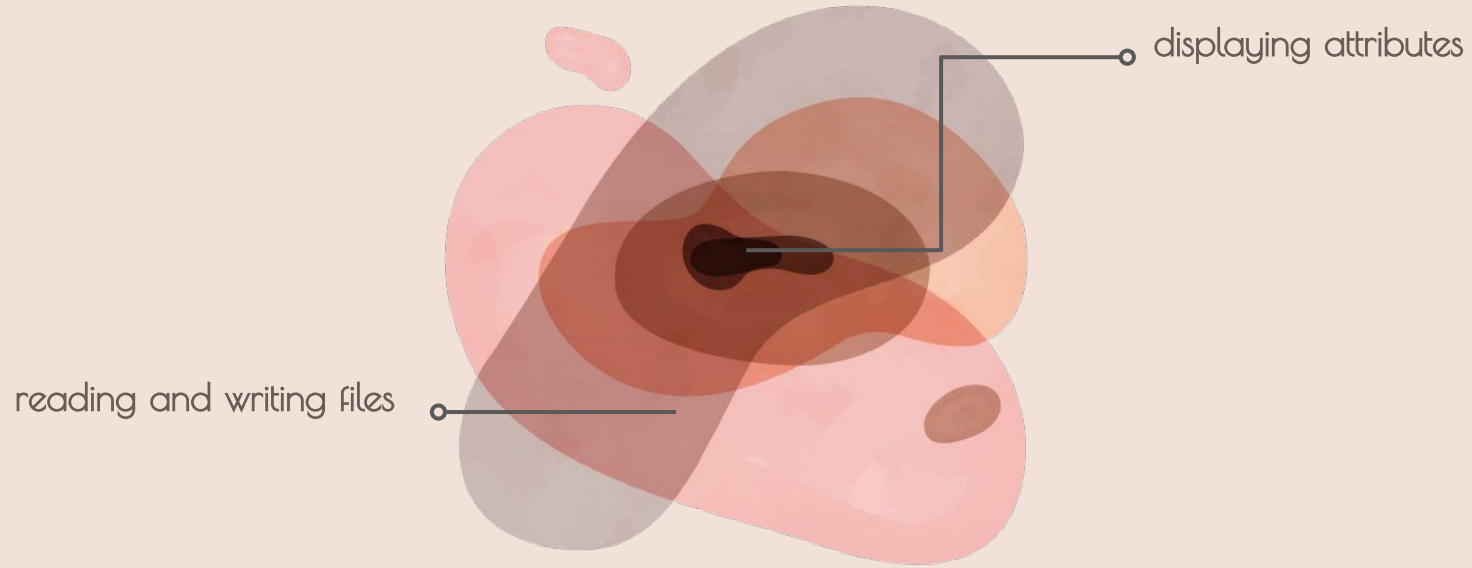
Writing to files was done twice, once for adoptability calculations and once for end of the day updates on the animals. This allowed for the easy creation of records, and is a lot more useful than simply printing it to a console. If this information was to be processed further, writing to files allows for its easy transportation (the next program can simply read the file, modify it, and use it). The format of a .txt file is also extremely easy to read for when you are just looking for the pure data.

Text files keep data separate from the data-processing code, making it more organized and easy to modify.



-  adoptabilities.txt
-  animal.py
-  bunnies.txt
-  bunny.py
-  cat.py
-  cats.txt
-  dog.py
-  dogs.txt
-  endOfDay.txt
-  pet.py

DATA CONVERSION



1) Reading and writing files

```
def stringToBool(x):  
    ...  
    Changes String values in the txt file to Boolean values  
  
    Parameters  
    -----  
    x: String  
    | the True or False value that is currently in String format  
  
    Returns  
    -----  
    boolean: bool  
    | will be true or false depending on x  
  
    ...  
    if x == "True":  
    | boolean = True  
    else:  
    | boolean = False  
    return boolean
```

With .txt files, everything is treated as a string in a list. When passing these lists as parameters, some parameters need to first be turned into their proper datatypes in order for the constructor to accept them.

Simple transformations such as string-to-int could be done with a one line function, which others such as string-to-boolean needed more complex algorithms. Algorithms were made into functions for easy repeated usage. By ensuring conversions are made correctly, we can ensure the program runs properly.

When writing back to .txt files, some variables need to be converted into a string in a similar fashion. In the example below, an array of Dog, Cat and Bunny objects need to be converted into one string before storing it in a .txt records page

```
age = int(age)  
childFriendly = int(childFriendly)  
gender = stringToBool(gender)  
forAdoption = stringToBool(forAdoption)  
neutered = stringToBool(neutered)  
scratches = stringToBool(scratches)
```

```
with open ('endOfDay.txt', 'w') as outputFile:  
    outputFile.write("DOGS" + str(dogs))  
with open ('endOfDay.txt', 'a') as outputFile:  
    outputFile.write("\n\nCATS" + str(cats))  
    outputFile.write("\n\nBUNNIES" + str(bunnies))
```

2) Displaying attributes

```
if self.gender == True:
    gender = "male"
elif self.gender == False:
    gender = "female"
```

```
if self.forAdoption == True:
    forAdoption = "yes"
else:
    forAdoption = "no"
if self.neutered == True:
    neutered = "yes"
else:
    neutered = "no"
```

When displaying attributes in the console or onto a .txt file, a lot of variables and filler words need to be concatenated. To do this, non-string variables need to be converted into strings. This allows all information to be gathered together and organized neatly before being stored/displayed. In the example below, the integer value `childFriendly` needed to be converted into a string.

With some conversions, the actual content was changed as well in order to make more logical output descriptions. For example, displaying "gender: True" does not make much sense. "True" is just a boolean used to represent male, so converting that boolean into a string makes for a meaningful conversion. It improves user experience while not changing the variable itself.

```
return "\n\n Name: " + self.name + "\n " + super().__str__() + "\n Breed:" +
self.breed + "\n For Adoption?: " + forAdoption + "\n Neutered?: " + neutered +
"\n Child Friendliness Rating: " + str(self.childFriendly)
```



end of walkthrough

HAPPY USING