# CSC 236 Data Structures

---

## Individual Assignment A05: Virtual Pets and ADTs

### Data Encapsulation Revisited

An abstract data type (ADT) is "a collection of functions or methods that manipulate an underlying representation", which is "just some collection of data." We first encountered this idea before when we worked with the Turtle library to create several objects of the type Turtle. Each turtle is an instance of the Turtle data type that has several attributes, such as orientation and whether its pen is down, and you used specific methods to change these states. For example, we called the `penup()` method to raise the pen so that the turtle does not draw anything while moving, and the `right()` method to change which way the turtle object is facing.

We next encountered the ADT idea when we learned to build classes.

The **`class`** mechanism is used in Python to define your own ADT. Suppose you wish to create a new Sedan class. The format would be:

```
class Sedan( object ):
```

All the methods associated with this class would be tab indented under this line. Some methods have special names, such as `__init__`, and all methods have a parameter typically called `self` in order to access the data of that object. For example, when creating an instance of a `Sedan`, it may make sense to start with no gas in the tank.. If you used a variable called `fuel_level` to represent the amount of gas in the tank, you could implement the constructor as:

```
    def __init__( self ):
        self.fuel_level = 0;
```

What kind of variable is `fuel_level`? It is a **`instance variable`** because (1) the value needs to be "remembered" by the object from one method call to another, and (2) the value is unique to a specific object (i.e. it is possible for there to be two `Sedan` objects where one has 20 gallons of gas and the other is empty.) One can recognize that it is an instance variable because it is only used inside an instance and it is preceded by "`self.` "

Suppose all sedans drive 20 miles per gallon. We can represent this as a **`class variable`** called `MPG`because ALL `Sedan` objects share that value. We can implement it as a variable declaration inside a class but NOT in a method:

```
class Sedan( object ):
    MPG = 20

    #methods follow
```

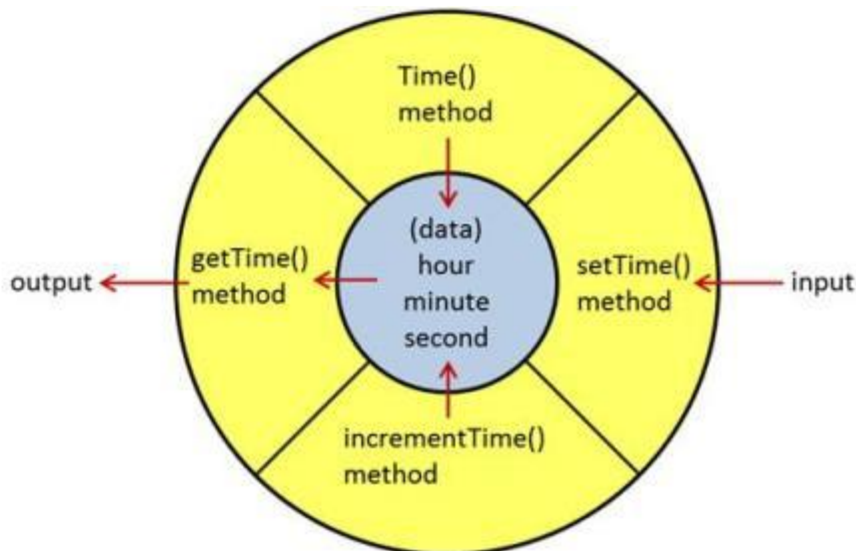
Finally, we have **`local variables`** that only exist inside of a specific method. For example, if we have a function called `drive` that has `num_miles` as a parameter, we can decrease the fuel in the tank using a local variable called `fuel_used`.

```
def drive(self,num_miles):
    fuel_used = num_miles / 20
    fuel_level = fuel_level = fuel_used
```

What makes a data structure an ADT instead of just a class? It is how the client program interacts with the data--with an ADT, you use methods to access the class data, never accessing it directly without using a method.

Let's see a plan for an ADT

| **Class name:** | **Sedan** |
|---|---|
| **Class Responsibilities (data and/or methods):** | **Class Collaborations (other classes):** |
| • data: remember MPG (classwide)<br>• data: remember amount of fuel<br>• method: be able to drive some # miles | •<br>•<br>• |




The only way to access the data is through these methods, and the details of HOW the data is manipulated or even how the data is represented is hidden from view.

Notice how the input and output operations on the data in the picture to the right is restricted to the methods.

There are no other ways for the program to access `fuel_level`, so it is possible to change the representation of that number (perhaps from integer to decimal) without impacting the way that the program works with the object. All that needs changing is the way that the methods that manipulate that number are implemented.

**The Task**

**This assignment is to be started in teams during class when you design the classes, but each person will implement their own version outside of class period.**

You are to create a program in which the user interacts with virtual pets with different needs. See **A05: Virtual Pets** for the description of the implementation which you will do individually.

A virtual pet can have, for example, three properties such as (1) the amount of time before it needs to be fed again, (2) its anxiety level, and (3) the amount of time the pet can stand to be alone. Each of these properties change depending on which methods of the pet class are called.

We welcome you to be creative when designing this program with your group, but your program must have the following:

1. there must be multiple classes - remember that classes typically are the nouns in object oriented design.
2. object instances must interact with each other through their methods, which are typically verbs in object oriented design.
3. all methods must work on ONE operation; try not to create multipurpose functions.
4. be sure to make appropriate use of local variables, instance variables, and class variables.
5. be sure to document your code appropriately both with docstrings and within inline comments.
6. have fun with this!

You may find the following program with the `Compressor` and `Hopper` classes interacting with each other helpful as an example.

Note that her in this first planning phase, the pre-conditions and post conditions are not yet determined, but you will find them in the code. It is typical to go back and forth as you design.

| **Class name:** | **Hopper** |
| --- | --- |
| **Class Responsibilities (data and/or methods):** | **Class Collaborations (other classes):** |
| • data: remember amount of coal stored<br>• method: be able to make new hopper<br>• method: be able to refill coal<br>• method: be able to use coal<br>• method: be able to print coal amount | •<br>•<br>•<br>•<br>• |

| **Class name:** | **Compressor** |
| --- | --- |
| **Class Responsibilities (data and/or methods):** | **Class Collaborations (other classes):** |
| • data: remember max capacity (classwide)<br>• data: remember max chances to make diamonds (classwide)<br>• data: remember number diamonds made<br>• data: remember number of compressions completed<br>• method: get goal from hopper<br>• method: compress coal and possibly create a diamond<br>• method: report out on coal used and diamonds created | •<br>•<br>•<br>•<br>• Hopper<br>•<br>• |

The main function is located in the *compressor_main.py* file and the other files are the class implementations:

1. **hopper.py**
2. **compressor.py**
3. **compressor_main.py**

**To submit**

You worked as a team when designing the classes needed for your program. Please design them as ADTs. Note that you do not need to put the pre-conditions and post-conditions in the design doc unless it is helpful to do so--they are required in the implementations.

You must implement and submit your program INDIVIDUALLY, though consultations with other people are allowed as long as they are acknowledged. You are also certainly welcome to change your design as you see fit as you implement your program.

1. Create a folder called *yourusername-A05*

2. Move all your implementation files (each module that defines each class you are using) into this folder.
3. Zip this directory and submit your zip-file, *yourusername-A05.zip*, onto Moodle when you are done.

---

Copyright © 2016 | Licensed under a Creative Commons Attribution-Share