CSE210 Notes

**Classes**

Classes are used in OOP. They don’t exist in Procedural languages

Procedural languages store data separately. No ability to store data within functions

OOP objects are classes with data and methods within them.

Functions become methods when in a class

Data and methods can be called and used in other classes

Instances can be created with objects. Ford = Car(), honda = Car() etc

Variables and methods can be created in instance with .method. ford.color = ‘red’

The pass keyword creates an empty class or method.

The def \_\_init\_\_(self): function acts LIKE a constructor, but it isn’t a constructor because it has automatic garbage collection

The \_\_init will run for every instance

Arguments can be passed into the class and then passed through in the instance

Class Car:

Def \_\_init\_\_(self, speed, color):

print(speed)

print(color)

The \_\_init function will print the passed through arguments in the functions but not create a variable to store the value. Self.attribute(variable) will create the variable/attribute

self.speed = speed

self.color = color

ford = Car(200, ‘blue’

Class attributes and instance attributes are separate entities and stored separately

**Self.variables in the init method are like global variables for that class**

Class Car():

Class\_var = 1

Def \_\_init\_\_(self, var\_):

Self.var\_ = var\_

Class\_var is the same for all instances, but var\_ is different according to instances.

**Object-Role Stereotypes**

**CRC**

* Modeling technique known as CRC cards **Class or Candidate/Responsibility/Collaborators**
* Use 3x5 card with simple sentence for purpose of class
* Divide lined side into two columns, left lists responsibility of class, right lists other class that must be interacted with to fulfill duties of the left side

**Responsibility-Driven Design**

* Breaking down to various actions and activities
* Consider which objects will be needed and how the responsibilities be distributed between objects

| **Stereotype** | **Description** |
| --- | --- |
| Information Holder | Knows things and provides information. May make calculations from the data that it holds. |
| Structurer | Knows the relationships between other objects. |
| Controller | Controls and directs the actions of other objects. Decides what other objects should do. |
| Coordinator | Reacts to events and relays the events to other objects. |
| Service Provider | Does a service for other objects upon request. |
| Interfacer | Objects that provide a means to communicate with other parts of the system, external systems or infrastructure, or end users. |

**Design around a behavior rather than designing from a data-centric viewpoint**

**Avoid the primitive obsession. Don’t be afraid to create small objects instead of wallowing in the mud of primitive variables**

**Structurer –** track, store, and maintain relationships between objects

* In many-to-many relationships amoung entities, will often need a structurer between everything
* Will be able to replace structurer if data grows in scope, rather than modifying other classes

**Service Provider** is usually passive – performs a task on behalf of another object

**Coordinator** is useful when you have a process that is event driven

**Laboratory for Teaching Object Orientation**

 We settled on three dimensions which identify the role of an object in a design: class name, responsibilities, and collaborators.

**Class names** – Important for design. Find the right words that are internally consistent with the design

**Responsibilites** identify problems to be solved. Solutions will exist in many versions and refinements

* Expressed in a handful of short verb phrases

**Collaborators –** No class or object is an island. Will send or be sent messages in the course of satisfying responsibilities

**ENCAPSULATION**

**Encapsulation** allows us to create private class attributes (variables) that are only modified and retrieved through a getter or setter class method

**Creating attribute**

* Def \_\_init\_\_(self):
  + Self.\_\_att = x
* Def att\_getter(self):
  + Return self.\_\_att
* Def att\_setter(self, value):
  + Self.\_\_att = value

**Setter (mutator)** – can be used for triggers when modifying data

**Getter(**accessor) – used for returning values

Def \_\_init\_\_(self, x):

Self.\_\_att = x

**Can be used for additional accessing if needed**

**PROPERTIES**

A getter method can be decorated with “@property”

No, it’s not --- A setter is decorated with “@x.setter” where x is the function name so “@(func\_name).setter”

**USE THIS INSTEAD** – attrib = property(fget, fset, fdel, doc)

* Fget = getter, fset = setter, fdel = delete (del c.attrib -> deletes)

The attrib attribute can be used to access the setter and getter methods

* C = Class(‘Bob’)
* Print(c.attrib) -> uses the getter method
* C.attrib = ‘Sam’
* Print(c.attrib) -> uses the setter method

**INHERITANCE**

Allows subclasses to inherit properties and methods from a superclass

Class Class\_1:

\_\_var1 = None

\_\_var2 = None

Class Class\_s2(Class\_1):

Etc

* Private variables cannot be accessed directly from subclass
* Need to create getter methods in order to access the private variables
  + Def get\_width(self) -> Class [different class] def function return self.get\_width()

**Super()**

Allows methods or properties? From superclass to be accessible in subclass

**BASICALLY CREATING AN OBJECT OF THE SUPERCLASS WITHIN THE SUBCLASS**

* **Can pass values into it like you would the object**
* **Super()\_\_init\_\_(‘Dog’) would be like superclass = Superclass(‘dog’)**

Class Square:

Def \_\_init\_\_

Super().\_\_init\_\_(not self parameters)

* Will now make all self.variables and non-private methods available?

Super takes two optional parameters: subclass, and an object

Super(class\_2, self):

Def class\_3 function:

Variable = super(class\_2, self).method()

* Will look above that subclass for the that inherited method?

**Use the parameter-less super() call normally**

**Subclasses can inherit from two super-classes (related or unrelated)**

Class subclass\_1(super\_1, super\_2):

Variable = super().super\_method()

Return variable

* Will look for methods in first parameter superclass first, so watch order or method names

**POLYMORPHISM**

Objects of different types can be accessed through the same interface

Subclasses methods and attributes of the same name will overwrite the superclass properties

If the superclass property wants to be accessed, an object instance of the superclass needs to be created and used.

**Method overloading**

Methods can be defined in a way that there are multiple ways to call it

Given a single method or function, we can specify the number of parameters ourselves

* Can use default parameters when defining function and have multiple options
  + Def sayHello(self, name=None):
    - If name is not None:
      * Print(“no name provided”)
    - Else:
      * Print(“Hello” + name)
  + Obj=Class()
  + Obj.sayHello() -> will execute first path

**Does a default parameter HAVE to be included in Method Overloading??**

**ABSTRACT CLASSES**

Prevents superclass from becoming an object on it’s own

* Superclass will act as a template for subclass only with no independent instances
* Any decorated (with the @abstractmethod) methods will make the class un-object-able and
* Can only be accessed through sub-class
* At least one method must be decorated with @abstractmethod

From abc import ABC, abstractmethod -> **ABC stands for Abstract Base Classes**

Class Shape(ABC):

@abstractmethod

Def function(self):pass

@abstractmethod

Def function (self):pass

**All the abstract methods must be instantiated in the subclass!**

So all methods with the @abstractmethod decorate must be re-declared in the subclass

**DUCK TYPING**