OBJECT ORIENTED PROGRAMMING

CS 3080: Python Programming

University of Colorado
Colorado Springs

What Is Object-Oriented Programming (OOP)?

- Is a **programming paradigm** which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
  - *For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running.*
- OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Classes in Python

- Each thing or object is an instance of some class.
- Think of a class as the idea of how something should be defined. A class provides structure! **It is a blueprint**.

- Let’s say you want to track a number of different animals and their age.
  - *List?* *[‘Deer’,8]* *What about more animals?*
    What about adding other properties?
    This lacks organization! And it is where we need classes.

- The Animal() class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal’s name or age is.

Python Objects (Instances)

- While the class is the blueprint, an **instance** is a copy of the class with actual values, literally an object belonging to a specific class.
- It’s not an idea anymore; it’s an actual animal, like a dog named Roger who’s eight years old.

Defining a class

```
class Dog:      # CamelCase notation starting with a capital letter
    pass                    # pass is used to create empty classes, methods,
                            # whiles, if, else, etc
```

You start with the class keyword to indicate that you are creating a class, then you add the name of the class (using CamelCase notation, starting with a capital letter.)

*Also, we used the Python keyword **pass** here. This is very often used as a place holder where code will eventually go. It allows us to run this code without throwing an error.*

Instance attributes

- The __init__ method is similar to constructors in C++ and Java.

  It is run as soon as an object of a class is instantiated. This method is useful to do any initialization you want to do with your object.

- All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph)
- Use the __init__() method to initialize (e.g. specify) an object’s initial attributes by giving them their default value (or state).
- This method must have at least one argument as well as the self variable, which refers to the object itself (e.g. Dog)
- The class is just for defining the Dog, not actually creating *instances* of individual dogs with specific names and ages.
- Similarly. the self variable is also an instance of the class. Since instances of a class have varying values, we could state Dog.name = name rather than self.name = name. But, since not all dogs share the same name, we need to be able to assign different values to different instances. Hence the need for the special self variable which will help to keep track of individual instances of each class.

Class attributes

Class attributes are the same for all instances when creating them.

Instantiating Objects

Instantiating is a fancy term for creating a new, unique instance of a class.

Instance methods

- Instance methods are defined inside a class and are used to **get the contents of an instance**.
- They can also be used to perform operations with the attributes of our objects.

Python Object Inheritance

- Inheritance is the process by which one class takes on the attributes and methods of another.
- Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.

Python Object Inheritance

We can **override** methods from the parent class:

Python Object Inheritance

Two ways to calling parent methods from a child class:

- *ParentClassName.method(self, arg1, arg2)* == *super().method(arg1, arg2)*

```
class Employee(Person):

    def __init__(self, first, last, staffnum):
        Person.__init__(self, first, last)
        self.staffnumber = staffnum

    def description(self):
        return super().description() + ", " +  self.staffnumber
```

issubclass()

isinstance()

Multiple inheritance

- Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as comma separated list in bracket.
- Python looks for each attribute in the class's parents as they are listed left to right
- For attributes, it will be the last __init__ called.

Encapsulation

- Encapsulation restrict access to methods and variables to prevent data from direct modification.
  - *Denote **private attributes** using single or double underscore as prefix.*

- Single or double underscore “ _ “ or “ __“.

- In the program, we defined a class Computer. We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can’t change it because Python treats the __maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

- Private methods are accessible outside their class, just not easily accessible. **Nothing in Python is truly private;**

Polymorphism

- Polymorphism means that different types respond to the same methods and attributes.
- Polymorphism is a fancy word that just means the same function or attribute is defined on objects of different types.
- This permits your code to use entities of different types at different times.
- Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.
- Polymorphism can be carried out through:
  - ***inheritance****, with subclasses making use of base class methods or overriding them.*
  - *Having the same method names in several classes or subclasses with different implementations*
    - Because the classes are polymorphic because they are using a single interface to use with entities of different types