Python OOP Session 7

Object-Oriented Programming (OOP)

- OOP is a method of structuring a program by bundling related properties and behaviors into individual objects
- Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects
- OOP models real-world entities as software objects that have some data associated with them and can perform certain functions

Define a Class in Python

Primitive data structures like numbers, strings, and lists are designed to represent simple pieces of information

Let's you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.
One way to do this is to represent each employee as a:

```
# Store employee info
employee1 = ["employee1", 34, "Manager", 2000]
employee2 = ["employee2", 35, "Science Officer", 2005]
employee3 = ["employee3", "Chief Medical Officer", 2010]
```

Classes vs Instances

- Classes are used to create user-defined data structures
- Classes define functions called methods
- Methods identify the behaviors and actions that an object created from the class can perform with its data

- A class is like a form or questionnaire. An instance is like a form that has been filled out with information
- A class is a blueprint for how something should be defined. It doesn’t actually contain any data. An instance is an object that is built from a class and contains real data

How to Define a Class

- Start with **class** keyword
- Followed by the name of the class and a colon
- Any code that is indented below the class definition is considered part of the class’s body

```
# Define class ex.
class DogAnimal:
    pass
```

Note

- The pass keyword. pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error
- Python class names are written in CapitalizedWords notation by convention
- Properties that all Dog objects must have are defined in a method called .**init**()
- Every time a new Dog object is created, .**init**() sets the initial <b> state</b> of the object by assigning the values of the object’s properties
- .**init**() initializes each new instance of the class
- can give .**init**() any number of parameters, but the first parameter will always be a variable called self
- When a new class instance is created, the instance is automatically passed to the self parameter in .**init**() so that new attributes can be defined on the object

```
# Update Dog class with init
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- self.name= name creates an attribute called name and assigns to it the value of the name parameter.
- self.age = age creates an attribute called age and assigns to it the value of the age parameter.
- **init**() are called instance attributes. An instance attribute's value is specific to a particular instance of the class
- class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of .**init**()

```
class Dog:
    # Class attribute
    Dogs_place = "place"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- Class attributes must always be assigned an initial value
- When an instance of the class is created, class attributes are automatically created and assigned to their initial values
- Use class attributes to define properties that should have the same value for every class instance
- Use instance attributes for properties that vary from one instance to another

Instantiate an Object in Python

Creating a new object from a class is called <b>instantiating</b> an object

```
# Define simple class
class Dog:
     pass
```

```
Dog()
```

- Creating a new object from a class is called instantiating an object
- String of letters and numbers is a memory address that indicates where the Dog object is stored

```
# second Dog object
Dog()
```

The new Dog instance is located at a different memory address

```
# Try to check two instances
a = Dog() # instance 1
b = Dog() # instance 2
a == b
```

Class and Instance Attributes

```
# Define class with attributes
class Dog:
     # class attribute
     dogs_place = "place"
```

```
    # Instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
# try to create instance
Dog()
```

- You need to provide values for the name and age
- To pass arguments to the name and age parameters, put values into the parentheses after the class name

```
# Create two instances
dog1 = Dog("dog1", 9)
dog2 = Dog("dog2", 4)
```

```
# Retrieve attributes
print(dog1.name,dog1.age,dog1.dogs_place)
print(dog2.name,dog2.age,dog2.dogs_place)
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect

```
# Change attributes
dog1.age=15
print(dog1.age)
# class attribute
dog1.dogs_place='place2'
print(dog1.dogs_place)
```

Custom objects are mutable by default. An object is mutable if it can be altered dynamically

Instance Methods

- Functions that are defined inside a class and can only be called from an instance of that class
- Instance method's first parameter is always self

```
# Define class with attributes and methods
class Dog:
     # class attribute
    dogs_place = "place"
     # Instance attribute
    def __init__(self, name, age):
         self.name = name
         self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

This Dog class has two instance methods:

- .description() returns a string displaying the name and age of the dog.
- .speak() has one parameter called sound and returns a string containing the dog's name and the sound the dog makes.