Object Oriented Programming in Python:

* OOP is a programming paradigm that revolves around the creation of classes and objects.
* The purpose is to bind the data and the functions that work around the data together as a single unit.
* Class:
  + A blueprint for creating new objects.
  + You define the objects’ properties, methods, and other components.
* Object:
  + A runtime instance of a class.
  + Objects consist of three things: State, Behavior, and Identity.
    - State – the attributes of an object.
    - Behavior – the methods of an object.
    - Identity – the identifier (name) of the object.
* Attributes:
  + The variables that belong to a class.
  + They can be accessed using the dot operator (.)
  + They can be deleted using the “del” keyword.
* Self:
  + Class methods must have an additional parameter in the method definition in order to utilize the class’s attributes.
* \_\_init\_\_:
  + Runs as soon as an object of the class is instantiated.
* Inheritance:
  + When one class inherits attributes / methods from another class.
  + Written in Python like:
    - class MySubclass(ParentClass)
  + Super() is a function used when you want to refer to the method of a parent class.
    - This is commonly done with the \_\_init\_\_ method to avoid repeating code.
    - Ex. Square inherits from Rectangle:

class Rectangle

def \_\_init\_\_(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

def perimeter(self):

return 2\*self.length + 2\*self.width

class Square(Rectangle):

def \_\_init\_\_(self, length):

super().\_\_init\_\_(length,length)

* + - Super() takes two parameters: the subclass and an object that is an instance of that subclass.
    - The subclass argument is useful if you have multi-level inheritance and want to be sure that a certain method call doesn’t use the subclass implementation, but the parent class’s implementation.
  + Python supports Multiple Inheritance – inheriting from two classes at once.
  + Ex. class Child(Mother, Father):
  + .\_\_mro\_\_:
    - Method Resolution Order.
    - This is a built-in attribute of all classes that tells the system where to look for method calls.
    - When creating the class as Child(Mother, Father) the system will first look in the Mother parent class for the method and attempt to execute that method if it is found.
    - To change the MRO, change the order in which the parent classes appear in the ().
    - The smarter solution, however, is to be sure your methods are simply named differently.
* Encapsulation:
  + A way to wrap or ‘encapsulate’ data so that it is not accessible by code outside of the class.
  + Attributes have two main access modifiers: Private and Protected
    - Protected
      * Protected attributes can be accessed inside the class and in its subclasses.
      * In Python, make an attribute Protected by preceding it with an underscore. (\_)
      * In practice, this doesn’t actually prevent anything, but it is a general convention that we should follow.
    - Private
      * Private attributes can only be accessed by the class in which they are defined.
      * In Python, make an attribute Private by preceding it with a double underscore. (\_\_)
* Abstraction:
  + Abstraction is a way to create methods to be implemented by child classes.
  + Abstract methods are methods that have declaration but no implementation.
  + It is useful when you want to provide a common interface for different implementations of the component.
  + Abstraction is not part of Python inherently, it must be imported:
    - from abc import ABC, abstractmethod, abstractproperty

class Polygon(ABC):

@abstractmethod

def num\_sides(self):

pass

class Triangle(Polygon):

def num\_sides(self):

print(“I have 3 sides.”)

* + Abstract classes cannot be instantiated.
* Property:
  + Returns an object of the property class and it is used to create properties of a class.
  + It is basically a replacement for getter, setter, and delete methods.
  + property(fget, fset, fdel, doc)
    - fget – gets the value of the attribute
    - fset – sets the value of the attribute
    - fdel – deletes the attribute value
    - doc – contains the docstring for the attribute

class Geeks:

def \_\_init\_\_(self):

self.\_age = 0

def get\_age(self):

print(“Getter method called”)

return self.\_age

def set\_age(self, a):

print(“Setter method called.”)

self.\_age = a

def del\_age(self):

del self.\_age

age = property(get\_age, set\_age, del\_age, )

Mark = Geeks()

Mark.age = 10

print(Mark.age)

* \_\_dict\_\_:
  + A special bult-in attribute that gives a list of mutable attributes as key-value pairs.
  + \_\_dict\_\_ is available on classes, instances of classes, user-defined functions, and modules.
  + To obtain a specific attribute, you can use the dictionary notation:
    - Ex. person1.\_\_dict\_\_[‘name’]
  + You can also use this to add new attributes to objects or functions (not classes).
  + To delete attributes, simply use “del”
  + Difference from dir()
    - Dir() lists all attributes, taking into consideration all instances, its classes, and inherited classes.
    - \_\_dict\_\_ only lists the local attributes.
* \_\_slots\_\_:
  + A memory optimizing measure.
  + Reduces the size of objects by allocating space for a fixed number of attributes.
  + Ex.

**class** GFG(object):

      \_\_slots\_\_**=**['a', 'b']

**def** \_\_init\_\_(self, **\***args, **\*\***kwargs):

                self.a **=** 1

                self.b **=** 2

* + Using \_\_slots\_\_ will prohibit the use of \_\_dict\_\_.
* Name Mangling:
  + Recall from earlier: placing the double underscore (\_\_) is how we made an attribute “Private”.
  + What this actually does internally is change the name of the attribute.
  + Ex.

**class** Student:

**def** \_\_init\_\_(self, name):

        self.\_\_name **=** name

s1 **=** Student("Santhosh")

**print**(s1.\_Student\_\_name)

* + \_\_name is now \_Student\_\_name
  + The class name has been placed before the attribute name.
  + The attributes can be accessed and changed outside of the class by using this new name.
  + This can also be a way to make sure overriding methods doesn’t cause conflict:
  + Ex.

**class** Map:

**def** \_\_init\_\_(self):

        self.\_\_geek()

**def** geek(self):

**print**("In parent class")

    # private copy of original geek() method

    \_\_geek **=** geek

**class** MapSubclass(Map):

    # provides new signature for geek() but

    # does not break \_\_init\_\_()

**def** geek(self):

**print**("In Child class")

# Driver's code

obj **=** MapSubclass()

obj.geek()

* + Output:

>> obj = MapSubclass()

In Parent class

>> obj.geek()

In Child class

* Introspection:
  + A general term meaning to examine the code and determine properties of objects during runtime.
  + Examples of Introspection:
    - type()
    - dir()
    - str()
    - id()
    - help()
    - hasattr()
    - getattr()
    - repr()
    - callable()
    - issubclass()
    - isinstance()
    - sys()
    - \_\_doc\_\_
    - \_\_name\_\_
* Hasattr()
  + A built-in method that determines if an attribute exists for a given object and returns True or False.
* \_\_str\_\_:
  + This is a built-in method that returns a string representation of an object.
  + It can be overridden in class definitions and will be called when you print the object.
* \_\_bases\_\_:
  + This is a built-in method that shows the base classes that a class inherited from.
* \_\_class\_\_:
  + This is a built-in method that shows what class an object is instantiated from.
* Diamonds:
  + This is a scenario where two classes inherit from the same base class, and another subclass inherits from those two.
  + Ex.

class A:

def myprint(self):

print(“This is class A”)

class B(A):

def myprint(self):

print(“This is class B”)

class C(A):

def myprint(self):

(“This is class C”)

class D(B,C):

def myprint(self):

super().myprint()

* + When calling super() in class D, it will consult the MRO for which class to refer to.
    - super() will call the myprint method from class B.
    - super(B,self) will call the myprint method from class C.
    - super(C,self) will call the myprint method from class A.