1. Make a class called Thing with no contents and print it. Then, create an object called example from this class and also print it. Are the printed values the same or different?

Ans : Certainly! Here's an example of how you can create a class called Thing with no contents, create an object called example from this class, and print both the class and the object:

class Thing: pass print(Thing()) example = Thing() print(example)

Output:

<\_\_main\_\_.Thing object at 0xXXXXXXXX> <\_\_main\_\_.Thing object at 0xXXXXXXXX>

In this example, I defined a class called **Thing** with no contents using the **pass** statement. Then, I printed an instance of the class **Thing** by calling **print(Thing())**. Additionally, I created an object called **example** from the **Thing** class and printed it using **print(example)**.

The printed values for both the class and the object are similar and follow the format **<\_\_main\_\_.Thing object at 0xXXXXXXXX>**. However, the **0xXXXXXXXX** part represents the memory address where the object is stored, so the actual values would be different.

2. Create a new class called Thing2 and add the value 'abc' to the letters class attribute. Letters should be printed.

Ans : Certainly! Here's an example of how you can create a new class called Thing2 and add the value 'abc' to the letters class attribute. Then, you can print the letters class attribute:

class Thing2: letters = 'abc' print(Thing2.letters)

Output:

abc

In this example, I defined a class called **Thing2** with the **letters** class attribute set to the value **'abc'**. By accessing the class attribute **Thing2.letters**, you can directly print the value **'abc'**.

3. Make yet another class called, of course, Thing3. This time, assign the value 'xyz' to an instance (object) attribute called letters. Print letters. Do you need to make an object from the class to do this?

Ans : Certainly! Here's an example of how you can create another class called Thing3 and assign the value 'xyz' to an instance attribute called letters. Then, you can print the letters attribute. Yes, you need to create an object from the class in order to access and print the instance attribute:

class Thing3: def \_\_init\_\_(self): self.letters = 'xyz' example = Thing3() print(example.letters)

Output:

xyz

In this example, I defined a class called **Thing3** with an **\_\_init\_\_** method. The **\_\_init\_\_** method is a special method that is executed when an object is created from the class. Inside the **\_\_init\_\_** method, I assigned the value **'xyz'** to the instance attribute **self.letters**.

Then, I created an object called **example** from the **Thing3** class using **example = Thing3()**. By accessing the instance attribute **example.letters**, you can print the value **'xyz'**.

4. Create an Element class with the instance attributes name, symbol, and number. Create a class object with the values 'Hydrogen,' 'H,' and 1.

Ans : Certainly! Here's an example of how you can create an Element class with the instance attributes name, symbol, and number, and then create a class object with the values 'Hydrogen', 'H', and 1:

class Element: def \_\_init\_\_(self, name, symbol, number): self.name = name self.symbol = symbol self.number = number element\_obj = Element('Hydrogen', 'H', 1)

In this example, I defined a class called **Element** with an **\_\_init\_\_** method. The **\_\_init\_\_** method is a special method that is executed when an object is created from the class. Inside the **\_\_init\_\_** method, I defined the instance attributes **name**, **symbol**, and **number** and assigned them the values passed as arguments.

Then, I created an object called **element\_obj** from the **Element** class by providing the values 'Hydrogen', 'H', and 1 as arguments. The object **element\_obj** will have the instance attributes **name** as 'Hydrogen', **symbol** as 'H', and **number** as 1.

5. Make a dictionary with these keys and values: 'name': 'Hydrogen', 'symbol': 'H', 'number': 1. Then, create an object called hydrogen from class Element using this dictionary.

Ans : Here's an example of how you can create a dictionary with the keys 'name', 'symbol', and 'number', along with their respective values. Then, you can create an object called hydrogen from the Element class using this dictionary:

class Element: def \_\_init\_\_(self, name, symbol, number): self.name = name self.symbol = symbol self.number = number element\_dict = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1} hydrogen = Element(\*\*element\_dict)

In this example, I defined the **Element** class with an **\_\_init\_\_** method that accepts the parameters **name**, **symbol**, and **number**. Inside the **\_\_init\_\_** method, I assign the values of these parameters to the respective instance attributes **self.name**, **self.symbol**, and **self.number**.

Then, I created a dictionary called **element\_dict** with the keys 'name', 'symbol', and 'number', along with their respective values. The **\*\*** syntax is used to unpack the dictionary and pass its key-value pairs as arguments to the **Element** class.

Finally, I created an object called **hydrogen** from the **Element** class using the dictionary values. The **hydrogen** object will have the instance attributes **name** as 'Hydrogen', **symbol** as 'H', and **number** as 1.

6. For the Element class, define a method called dump() that prints the values of the object’s attributes (name, symbol, and number). Create the hydrogen object from this new definition and use dump() to print its attributes.

Ans : Here's an updated example of the Element class with a dump() method that prints the values of the object's attributes (name, symbol, and number). Then, the hydrogen object is created using this new definition, and the dump() method is called to print its attributes:

class Element: def \_\_init\_\_(self, name, symbol, number): self.name = name self.symbol = symbol self.number = number def dump(self): print("Name:", self.name) print("Symbol:", self.symbol) print("Number:", self.number) hydrogen = Element('Hydrogen', 'H', 1) hydrogen.dump()

Output:

Name: Hydrogen Symbol: H Number: 1

In this updated example, the **Element** class still has the **\_\_init\_\_** method to initialize the instance attributes. Additionally, a new method called **dump()** is defined within the class. The **dump()** method takes **self** as the first parameter (representing the object itself) and prints the values of the object's attributes using the **print()** function.

The **hydrogen** object is created using the updated **Element** class definition, and the **dump()** method is called on the **hydrogen** object to print its attributes: 'Hydrogen' for **name**, 'H' for **symbol**, and 1 for **number**.

7. Call print(hydrogen). In the definition of Element, change the name of method dump to \_\_str\_\_, create a new hydrogen object, and call print(hydrogen) again.

Ans : print(hydrogen) and modify the method name from dump() to \_\_str\_\_ within the Element class. We create a new hydrogen object and call print(hydrogen) again:

pythonCopy code

class Element: def \_\_init\_\_(self, name, symbol, number): self.name = name self.symbol = symbol self.number = number def \_\_str\_\_(self): return f"Name: {self.name}\nSymbol: {self.symbol}\nNumber: {self.number}" hydrogen = Element('Hydrogen', 'H', 1) print(hydrogen) new\_hydrogen = Element('New Hydrogen', 'NH', 2) print(new\_hydrogen)

Output:

vbnetCopy code

Name: Hydrogen Symbol: H Number: 1 Name: New Hydrogen Symbol: NH Number: 2

In this updated example, the **\_\_str\_\_** method within the **Element** class is modified to return a string representation of the object. This method is automatically called when we use the **print()** function on an instance of the class.

We create the **hydrogen** object using the updated **Element** class definition and then call **print(hydrogen)**, which triggers the **\_\_str\_\_** method. The **\_\_str\_\_** method returns a formatted string containing the attribute values of the object.

We also create a new **new\_hydrogen** object and call **print(new\_hydrogen)** to see the updated string representation based on the modified **\_\_str\_\_** method.

8. Modify Element to make the attributes name, symbol, and number private. Define a getter property for each to return its value.

Ans : updated version of the Element class where the attributes name, symbol, and number are made private, and getter properties are defined to access their values:

class Element: def \_\_init\_\_(self, name, symbol, number): self.\_name = name self.\_symbol = symbol self.\_number = number @property def name(self): return self.\_name @property def symbol(self): return self.\_symbol @property def number(self): return self.\_number

In this updated example, the attributes **name**, **symbol**, and **number** are prefixed with a single underscore **\_**, indicating that they are intended to be private. Although Python doesn't enforce strict privacy, it is a convention to indicate that these attributes are intended for internal use within the class.

To provide access to these private attributes, getter properties are defined using the **@property** decorator. Each getter property method is named after the attribute it retrieves (**name**, **symbol**, and **number**), and they simply return the corresponding private attribute value (**self.\_name**, **self.\_symbol**, and **self.\_number**, respectively).

With these getter properties defined, you can access the attribute values of an **Element** object using dot notation, like **element.name**, **element.symbol**, and **element.number**, while still keeping the attributes private.

9. Define three classes: Bear, Rabbit, and Octothorpe. For each, define only one method: eats(). This should return 'berries' (Bear), 'clover' (Rabbit), or 'campers' (Octothorpe). Create one object from each and print what it eats.

Ans: Here's an example of defining three classes: Bear, Rabbit, and Octothorpe. Each class has a single method eats() that returns the respective food they eat. Then, we create one object from each class and print what they eat:

class Bear: def eats(self): return 'berries' class Rabbit: def eats(self): return 'clover' class Octothorpe: def eats(self): return 'campers' # Create objects and print what they eat bear = Bear() rabbit = Rabbit() octothorpe = Octothorpe() print(bear.eats()) # Output: berries print(rabbit.eats()) # Output: clover print(octothorpe.eats()) # Output: campers

In this example, we define three classes: **Bear**, **Rabbit**, and **Octothorpe**. Each class has a single method **eats()** that returns the respective food they eat.

We then create objects **bear**, **rabbit**, and **octothorpe** from each class, respectively. Finally, we call the **eats()** method on each object and print the output to see what they eat: 'berries' for the bear, 'clover' for the rabbit, and 'campers' for the octothorpe.

10. Define these classes: Laser, Claw, and SmartPhone. Each has only one method: does(). This returns 'disintegrate' (Laser), 'crush' (Claw), or 'ring' (SmartPhone). Then, define the class Robot that has one instance (object) of each of these. Define a does() method for the Robot that prints what its component objects do.

Ans : defining the classes Laser, Claw, SmartPhone, and Robot as described, along with their respective methods:

class Laser: def does(self): return 'disintegrate' class Claw: def does(self): return 'crush' class SmartPhone: def does(self): return 'ring' class Robot: def \_\_init\_\_(self): self.laser = Laser() self.claw = Claw() self.smartphone = SmartPhone() def does(self): print("Laser does:", self.laser.does()) print("Claw does:", self.claw.does()) print("SmartPhone does:", self.smartphone.does())

In this example, we define the classes **Laser**, **Claw**, **SmartPhone**, and **Robot**. Each class has a single method called **does()** which returns the respective action.

The **Robot** class has an **\_\_init\_\_** method to initialize the instance variables **laser**, **claw**, and **smartphone** as objects of the **Laser**, **Claw**, and **SmartPhone** classes, respectively.

The **Robot** class also has its own **does()** method which calls the **does()** methods of its component objects (**laser**, **claw**, and **smartphone**). This method prints what each component object does.

Here's an example usage:

pythonCopy code

robot = Robot() robot.does()

Output:

Laser does: disintegrate Claw does: crush SmartPhone does: ring

In this example, we create an object **robot** of the **Robot** class. We then call the **does()** method on the **robot** object, which in turn calls the **does()** methods of its component objects (**laser**, **claw**, and **smartphone**). The output shows what each component object does: 'disintegrate' for the laser, 'crush' for the claw, and 'ring' for the smartphone.