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: The `Thing` class with no contents, instantiate an object called `example` from this class, and print both the class and the object to see if the printed values are the same or different.

class Thing:

pass

# Print the class

print(Thing)

# Create an object from the class

example = Thing()

# Print the object

print(example)

When you run the code, you'll get the following output:

<class '\_\_main\_\_.Thing'>

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

The printed values for the class and the object are different.

- The first line `<class '\_\_main\_\_.Thing'>` represents the class `Thing`. It shows the class type and the module it belongs to (`\_\_main\_\_` in this case, which is the main module when executing code interactively or from a script).

- The second line `<\_\_main\_\_.Thing object at 0x...>` represents the object `example` created from the `Thing` class. It shows the object type (`\_\_main\_\_.Thing`) and a memory address (`0x...`) where the object is stored in memory.

Therefore, the printed values are different. The first line represents the class `Thing`, and the second line represents the object `example` created from the `Thing` class.

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

ANS: The `Thing2` class and add the value 'abc' to the `letters` class attribute. Then, we'll print the value of the `letters` class attribute.

class Thing2:

letters = 'abc'

# Print the value of the letters class attribute

print(Thing2.letters)

When you run the code, it will print the value of the `letters` class attribute:

abc

The value 'abc' is stored in the `letters` class attribute of the `Thing2` class. We access class attributes using the class name followed by the attribute name (e.g., `Thing2.letters`).

1. 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: The `Thing3` class and assign the value 'xyz' to an instance attribute called `letters`. Then, we'll print the value of the `letters` attribute. To access the instance attribute, we indeed need to create an object (instance) of the class.

class Thing3:

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

self.letters = 'xyz'

# Create an object (instance) of the class

example = Thing3()

# Print the value of the 'letters' instance attribute

print(example.letters)

When you run the code, it will print the value of the `letters` instance attribute:

Xyz

In this case, the value 'xyz' is stored in the `letters` attribute of the `example` object (instance) created from the `Thing3` class. We access instance attributes using the object (instance) name followed by the attribute name (e.g., `example.letters`).

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

ANS: The `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

# Create a class object with the values 'Hydrogen,' 'H,' and 1

hydrogen = Element('Hydrogen', 'H', 1)

In this example, we defined the `Element` class with an `\_\_init\_\_` method that takes three arguments (`name`, `symbol`, and `number`) and assigns them to the corresponding instance attributes (`self.name`, `self.symbol`, and `self.number`).

Then, we created an object called `hydrogen` of the `Element` class and passed the values 'Hydrogen,' 'H,' and 1 as arguments to the `\_\_init\_\_` method.

Now, the `hydrogen` object has the instance attributes `name`, `symbol`, and `number` with the specified values. You can access these attributes as follows:

print(hydrogen.name) # Output: Hydrogen

print(hydrogen.symbol) # Output: H

print(hydrogen.number) # Output: 1

Each attribute of the `hydrogen` object holds the respective value we assigned during object creation.

1. 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: To create a dictionary with the given keys and values and then use this dictionary to create an object called `hydrogen` from the `Element` class, you can follow these steps:

class Element:

def \_\_init\_\_(self, name, symbol, number):

self.name = name

self.symbol = symbol

self.number = number

# Create a dictionary with the keys and values

element\_dict = {

'name': 'Hydrogen',

'symbol': 'H',

'number': 1

}

# Create an object 'hydrogen' from the 'Element' class using the dictionary

hydrogen = Element(\*\*element\_dict)

In this example, we first defined the `Element` class with the `\_\_init\_\_` method that takes three arguments (`name`, `symbol`, and `number`) and assigns them to the corresponding instance attributes (`self.name`, `self.symbol`, and `self.number`).Then, we created a dictionary named `element\_dict` with the keys `'name'`, `'symbol'`, and `'number'`, and their respective values `'Hydrogen'`, `'H'`, and `1`.Finally, we used the double asterisks (`\*\*`) notation to pass the dictionary as keyword arguments when creating the `hydrogen` object. The dictionary keys are mapped to the `\_\_init\_\_` method arguments based on their names.

Now, the `hydrogen` object has the instance attributes `name`, `symbol`, and `number` with the values obtained from the dictionary. You can access these attributes as shown in the previous response.

1. 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: A method called `dump()` inside the `Element` class that prints the values of the object's attributes (`name`, `symbol`, and `number`). Then, we'll create the `hydrogen` object using the updated `Element` class definition and use the `dump()` method 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)

# Create the hydrogen object from the Element class

hydrogen = Element('Hydrogen', 'H', 1)

# Use the dump() method to print its attributes

hydrogen.dump()

When you run the code, it will create the `hydrogen` object and then call the `dump()` method to print its attributes:

Name: Hydrogen

Symbol: H

Number: 1

The `dump()` method in the `Element` class prints the values of the object's attributes `name`, `symbol`, and `number`, giving us the desired output.

1. 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: Sure! Let's call `print(hydrogen)` before and after changing the method name to `\_\_str\_\_` in the `Element` class.

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}, Symbol: {self.symbol}, Number: {self.number}"

# Create the hydrogen object from the Element class

hydrogen = Element('Hydrogen', 'H', 1)

# Call print(hydrogen) before changing the method name

print(hydrogen)

# Change the method name from dump to \_\_str\_\_

# Create a new hydrogen object

hydrogen = Element('Hydrogen', 'H', 1)

# Call print(hydrogen) after changing the method name to \_\_str\_\_

print(hydrogen)

Output:

<\_\_main\_\_.Element object at 0x...>

Name: Hydrogen, Symbol: H, Number: 1

Before changing the method name to `\_\_str\_\_`, when you call `print(hydrogen)`, it prints the default representation of the object, which includes the object type and memory address (`<\_\_main\_\_.Element object at 0x...>`).After changing the method name to `\_\_str\_\_`, when you call `print(hydrogen)`, it prints the string representation returned by the `\_\_str\_\_` method, which displays the attribute values in the desired format.By defining the `\_\_str\_\_` method, you can control how the object is represented as a string when using the `print()` function or other string operations.

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

ANS: To make the attributes `name`, `symbol`, and `number` private in the `Element` class, we can use double underscores (`\_\_`) before their names. This convention in Python makes the attributes private and prevents direct access from outside the class. To access the private attributes, we'll define getter properties for each of them.

Here's the modified `Element` class with private attributes and getter properties:

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

# Create the hydrogen object from the Element class

hydrogen = Element('Hydrogen', 'H', 1)

# Use the getter properties to access the private attributes

print("Name:", hydrogen.name)

print("Symbol:", hydrogen.symbol)

print("Number:", hydrogen.number)

Output:

Name: Hydrogen

Symbol: H

Number: 1

In this modified class, the attributes `\_\_name`, `\_\_symbol`, and `\_\_number` are now private, and we use getter properties to access their values from outside the class. The getter properties allow us to retrieve the values of the private attributes without exposing them directly.

1. 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: The three classes `Bear`, `Rabbit`, and `Octothorpe`, each with a method `eats()` that returns the appropriate food for each animal. Then, we'll create one object from each class and print what each animal eats.

class Bear:

def eats(self):

return 'berries'

class Rabbit:

def eats(self):

return 'clover'

class Octothorpe:

def eats(self):

return 'campers'

# Create objects from each class

bear\_obj = Bear()

rabbit\_obj = Rabbit()

octothorpe\_obj = Octothorpe()

# Print what each animal eats

print("Bear eats:", bear\_obj.eats())

print("Rabbit eats:", rabbit\_obj.eats())

print("Octothorpe eats:", octothorpe\_obj.eats())

Output:

Bear eats: berries

Rabbit eats: clover

Octothorpe eats: campers

In this code, we defined the `Bear`, `Rabbit`, and `Octothorpe` classes, each with an `eats()` method that returns the corresponding food for each animal. We then created one object from each class (`bear\_obj`, `rabbit\_obj`, and `octothorpe\_obj`) and printed what each animal eats using the `eats()` method.

1. 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: The classes `Laser`, `Claw`, and `SmartPhone`, each with a method `does()` that returns the specified action. Then, we'll define the class `Robot` that has one instance (object) of each of these classes. In the `Robot` class, we'll define a `does()` method that prints what its component objects do.

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:", self.laser.does())

print("Claw:", self.claw.does())

print("SmartPhone:", self.smartphone.does())

# Create a Robot object

robot = Robot()

# Call the does() method for the Robot to print what its components do

robot.does()

Output:

Laser: disintegrate

Claw: crush

SmartPhone: ring

In this code, we defined the classes `Laser`, `Claw`, and `SmartPhone`, each with a `does()` method that returns the specified action. Then, we created a `Robot` class with one instance (object) of each of these classes as its attributes (`laser`, `claw`, and `smartphone`). The `does()` method of the `Robot` class calls the `does()` methods of its component objects and prints what each component does.