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?

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

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?

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

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.

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.

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

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.

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.

Answer:

1. Here's the implementation for the first part:

```python

class Thing:

pass

print(Thing())

example = Thing()

print(example)

```

The printed values will be different because the first print statement outputs the class name followed by its memory address, while the second print statement outputs the same but for the object `example` created from the class `Thing`.

2. Here's the implementation for the second part:

```python

class Thing2:

letters = 'abc'

print(Thing2.letters)

```

The output will be `abc` because `letters` is a class attribute, and it can be accessed directly using the class name `Thing2`.

3. Here's the implementation for the third part:

```python

class Thing3:

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

self.letters = 'xyz'

print(Thing3().letters)

```

You don't need to create an object explicitly to access the instance attribute `letters`. The code creates an instance of `Thing3` using `Thing3()` and then accesses the `letters` attribute using `.letters`.

4. Here's the implementation for the fourth part:

```python

class Element:

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

self.name = name

self.symbol = symbol

self.number = number

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

```

An object `element` of class `Element` is created with the values `'Hydrogen'`, `'H'`, and `1` assigned to the respective attributes `name`, `symbol`, and `number`.

5. Here's the implementation for the fifth part:

```python

class Element:

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

self.name = name

self.symbol = symbol

self.number = number

element\_data = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1}

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

```

The dictionary `element\_data` is used to pass the values to the `Element` class constructor. The `\*\*` operator is used to unpack the dictionary and pass the values as keyword arguments to the constructor.

6. Here's the implementation for the sixth part:

```python

class Element:

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

self.name = name

self.symbol = symbol

self.number = number

def dump(self):

print(f"Name: {self.name}, Symbol: {self.symbol}, Number: {self.number}")

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

hydrogen.dump()

```

The `dump()` method is defined within the `Element` class to print the values of the object's attributes `name`, `symbol`, and `number`. The `dump()` method is then called on the `hydrogen` object to print its attributes.

7. Here's the implementation for the seventh part:

```python

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}"

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

print(hydrogen)

```

The `dump()` method is renamed to `\_\_str\_\_()` within the `Element` class. This special method allows us to define a string representation of the object. The `hydrogen` object is then printed using `print(hydrogen)`, which internally calls the `\_\_str\_\_()` method.

8. Here's the implementation for the eighth part:

```python

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

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

print(hydrogen.name) # Accessing private attribute via getter property

print(hydrogen.symbol)

print(hydrogen.number)

```

The attributes `name`, `symbol`, and `number` are made private by prefixing them with `\_\_`. Getter properties are defined using the `@property` decorator to allow accessing the private attributes outside the class.

9. Here's the implementation for the ninth part:

```python

class Bear:

def eats(self):

return 'berries'

class Rabbit:

def eats(self):

return 'clover'

class Octothorpe:

def eats(self):

return 'campers'

bear = Bear()

rabbit = Rabbit()

octothorpe = Octothorpe()

print(bear.eats())

print(rabbit.eats())

print(octothorpe.eats())

```

The `Bear`, `Rabbit`, and `Octothorpe` classes are defined with a single method `eats()` that returns the food preference for each animal. Objects are created from each class, and the `eats()` method is called on each object to print their respective food preferences.

10. Here's the implementation for the tenth part:

```python

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(f"Laser: {self.laser.does()}")

print(f"Claw: {self.claw.does()}")

print(f"SmartPhone: {self.smartphone.does()}")

robot = Robot()

robot.does()

```

The `Laser`, `Claw`, and `SmartPhone` classes are defined with a single method `does()` that returns the action performed by each component. The `Robot` class is defined with instances of these component classes. The `does()` method in the `Robot` class is defined to print the actions performed by each component. An object `robot` is created from the `Robot` class, and its `does()` method is called to print the actions.