**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?**

Yes, the printed values for the class and the object are different.

Here's the code:

```python

class Thing:

pass

print(Thing)

example = Thing()

print(example)

```

Output:

```

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

<\_\_main\_\_.Thing object at 0x7f90f4b31850>

```

As we can see, the first print statement prints the class object, while the second print statement prints the object instantiated from the class.

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

Here is the code to create a new class called `Thing2` and add the value `'abc'` to the `letters` class attribute:

```python

class Thing2:

letters = 'abc'

print(Thing2.letters)

```

When we run this code, it will output:

```

abc

```

This is because we accessed the `letters` attribute of the `Thing2` class using the dot notation (`Thing2.letters`).

**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?**

Yes, we need to make an object from the class to access and print the instance attribute. Here's an example implementation:

```python

class Thing3:

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

self.letters = 'xyz'

example = Thing3()

print(example.letters)

```

Output:

```

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.**

Here's how you can create an `Element` class with the given instance attributes and create an object with the specified values:

```python

class Element:

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

self.name = name

self.symbol = symbol

self.number = number

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

```

This creates an `Element` class with an `\_\_init\_\_()` method that takes in three arguments: `name`, `symbol`, and `number`. These arguments are assigned to instance attributes of the same name using the `self` keyword.

The `element` object is then created by instantiating the `Element` class and passing in the values 'Hydrogen,' 'H,' and 1 as arguments. These values are assigned to the instance attributes of the `element` object.

**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.**

Here's how you can create an object called `hydrogen` from class `Element` using the given dictionary:

```python

class Element:

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

self.name = name

self.symbol = symbol

self.number = number

hydrogen\_dict = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1}

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

print(hydrogen.name)

print(hydrogen.symbol)

print(hydrogen.number)

```

Output:

```

Hydrogen

H

1

```

In this example, `\*\*hydrogen\_dict` unpacks the `hydrogen\_dict` dictionary and passes its key-value pairs as arguments to the `Element` constructor, which creates the `hydrogen` object. Then, the `print()` statements print the values of the `name`, `symbol`, and `number` attributes of `hydrogen`.

**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.**

Here's how we can define the Element class and its `dump()` method:

```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}")

```

Now, we can create the `hydrogen` object and use the `dump()` method to print its attributes:

```python

hydrogen\_dict = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1}

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

hydrogen.dump() # prints "name: Hydrogen, symbol: H, number: 1"

```

Note that we're using the dictionary `\*\*` operator to unpack the key-value pairs in the `hydrogen\_dict` dictionary and pass them as arguments to the `Element` constructor.

**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.**

Sure, here's an example implementation of the Element class with the `\_\_str\_\_` method:

```python

class Element:

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

self.name = name

self.symbol = symbol

self.number = number

def \_\_str\_\_(self):

return f"Element({self.name}, {self.symbol}, {self.number})"

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

print(hydrogen) # will print: Element(Hydrogen, H, 1)

```

In the above code, the `\_\_str\_\_` method is defined to return a string representation of the `Element` object. This allows us to call `print(hydrogen)` and have it print a human-readable representation of the object. The `\_\_str\_\_` method is automatically called when an object is passed to the `print()` function or when we try to convert the object to a string using `str()`.

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

To make the attributes `name`, `symbol`, and `number` private, we can add double underscores before their names (`\_\_name`, `\_\_symbol`, `\_\_number`). Then, we can define getter properties for each attribute using the `@property` decorator. Here's the modified `Element` class:

```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

```

Now, we can create an `Element` object and use the getter properties to access its private attributes. Here's an example:

```python

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

print(hydrogen.name) # Output: Hydrogen

print(hydrogen.symbol) # Output: H

print(hydrogen.number) # Output: 1

```

**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.**

Here is an example implementation of the three classes and the creation of their objects:

```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('The bear eats', bear.eats())

print('The rabbit eats', rabbit.eats())

print('The octothorpe eats', octothorpe.eats())

```

The output of this program will be:

```

The bear eats berries

The rabbit eats clover

The octothorpe eats campers

```

**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.**

Here's an implementation of the classes `Laser`, `Claw`, `SmartPhone`, and `Robot` as described in the question:

```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):

return f'Laser does {self.laser.does()}, Claw does {self.claw.does()}, SmartPhone does {self.smartphone.does()}'

```

We create an object of the `Robot` class and call its `does()` method to see what its component objects do:

```python

robot = Robot()

print(robot.does()) # Laser does disintegrate, Claw does crush, SmartPhone does ring

```

As expected, the output shows what each component object does.