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?**

Sure, here's a Python code snippet to create the `Thing` class, print it, and then create an object called `example` from this class and also print it:

```python

class Thing:

pass

print(Thing())

example = Thing()

print(example)

```

When you run this code, you will get output like the following:

```

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

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

```

The printed values will be different. Both lines indicate that you have an object of the `Thing` class, but the memory address (the hexadecimal value) will be different for each object. Each object gets its own unique memory location, even if they belong to the same class.

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

Sure, here's a Python code snippet to create the `Thing2` class with a class attribute `letters` set to the value `'abc'` and then print the value of the `letters` attribute:

```python

class Thing2:

letters = 'abc'

print(Thing2.letters)

```

When you run this code, you will get the following output:

```

abc

```

The `letters` attribute is a class attribute, meaning it belongs to the class `Thing2`, not to any specific instance of the class. Therefore, you can access it directly using the class name (`Thing2`) followed by the attribute name (`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, to access and print the instance attribute `letters` of the `Thing3` class, you need to create an object (instance) of the class. Here's the Python code to create the `Thing3` class with an instance attribute `letters` set to the value `'xyz'` and then print the value of the `letters` attribute:

```python

class Thing3:

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

self.letters = 'xyz'

# Create an object (instance) of the class

example = Thing3()

# Print the instance attribute 'letters'

print(example.letters)

```

When you run this code, you will get the following output:

```

xyz

```

In this case, the `letters` attribute is specific to the `example` object, so you need to create an instance of the class to access and print this attribute.

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

Yes, to access and print the instance attribute `letters` of the `Thing3` class, you need to create an object (instance) of the class. Here's the Python code to create the `Thing3` class with an instance attribute `letters` set to the value `'xyz'` and then print the value of the `letters` attribute:

```python

class Thing3:

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

self.letters = 'xyz'

# Create an object (instance) of the class

example = Thing3()

# Print the instance attribute 'letters'

print(example.letters)

```

When you run this code, you will get the following output:

```

xyz

```

In this case, the `letters` attribute is specific to the `example` object, so you need to create an instance of the class to access and print this attribute.

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.**

Sure, to create an object called `hydrogen` from the `Element` class using the given dictionary, you can use the dictionary unpacking feature in Python. Here's how you can do it:

```python

class Element:

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

self.name = name

self.symbol = symbol

self.number = number

# Create the dictionary with the keys and values for the attributes

hydrogen\_data = {

'name': 'Hydrogen',

'symbol': 'H',

'number': 1

}

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

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

# Access the attributes of the 'hydrogen' object

print(hydrogen.name) # Output: Hydrogen

print(hydrogen.symbol) # Output: H

print(hydrogen.number) # Output: 1

```

In this code, the `\*\*hydrogen\_data` syntax is used to unpack the dictionary `hydrogen\_data` and pass its key-value pairs as arguments to the `Element` class constructor (`\_\_init\_\_` method). This allows you to create the `hydrogen` object with the specified attribute values from the dictionary.

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.**

Sure, let's define the `dump()` method in the `Element` class, which will print the values of the object's attributes (`name`, `symbol`, and `number`):

```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}")

print(f"Symbol: {self.symbol}")

print(f"Number: {self.number}")

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

hydrogen\_data = {

'name': 'Hydrogen',

'symbol': 'H',

'number': 1

}

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

# Use the dump() method to print its attributes

hydrogen.dump()

```

When you run this code, you will get the following output:

```

Name: Hydrogen

Symbol: H

Number: 1

```

The `dump()` method allows you to print the values of the object's attributes in a structured manner. It provides a convenient way to inspect the attributes of an object.

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.**

When you call `print(hydrogen)`, Python internally calls the `\_\_str\_\_` method of the `hydrogen` object (if it exists) to convert the object to a string representation. By default, the `\_\_str\_\_` method returns a string representation of the object, but we can customize it to display the desired information.

Let's modify the `Element` class to change the method `dump` to `\_\_str\_\_`, and then create a new `hydrogen` object and call `print(hydrogen)` to see the updated behavior:

```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}\nSymbol: {self.symbol}\nNumber: {self.number}"

# Create a new 'hydrogen' object from the 'Element' class using the dictionary

hydrogen\_data = {

'name': 'Hydrogen',

'symbol': 'H',

'number': 1

}

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

# Call print(hydrogen) to see the string representation returned by \_\_str\_\_

print(hydrogen)

```

Output:

```

Name: Hydrogen

Symbol: H

Number: 1

```

By changing the method name to `\_\_str\_\_`, we can now directly use `print(hydrogen)` to get a nicely formatted string representation of the `hydrogen` object. The `\_\_str\_\_` method allows us to define how the object should be represented as a string when using the `print` function or implicitly converting the object to a string.

1. **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 in the `Element` class, we can prefix them with double underscores (e.g., `\_\_name`, `\_\_symbol`, `\_\_number`). This will effectively make them private and prevent direct access from outside the class. To provide controlled access to these private attributes, we can define getter properties using the `@property` decorator.

Let's modify the `Element` class accordingly:

```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

# Create a new 'hydrogen' object from the 'Element' class using the dictionary

hydrogen\_data = {

'name': 'Hydrogen',

'symbol': 'H',

'number': 1

}

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

# Access the private attributes using the getter properties

print(hydrogen.name) # Output: Hydrogen

print(hydrogen.symbol) # Output: H

print(hydrogen.number) # Output: 1

# Attempting to access private attributes directly will raise an AttributeError

# Uncomment the following lines to test it

# print(hydrogen.\_\_name)

# print(hydrogen.\_\_symbol)

# print(hydrogen.\_\_number)

```

In this modified `Element` class, the attributes `\_\_name`, `\_\_symbol`, and `\_\_number` are private, and you cannot access them directly from outside the class. Instead, you use the getter properties `name`, `symbol`, and `number` to retrieve their values, providing controlled access to these attributes. Any attempt to access the private attributes directly will raise an `AttributeError`.

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.**

Sure, let's define the `Bear`, `Rabbit`, and `Octothorpe` classes with the `eats()` method, each returning the respective food they eat: `'berries'`, `'clover'`, and `'campers'`. Then, we'll create one object from each class and print what they eat:

```python

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 = Bear()

rabbit = Rabbit()

octothorpe = Octothorpe()

# Print what each object eats

print("Bear eats:", bear.eats()) # Output: Bear eats: berries

print("Rabbit eats:", rabbit.eats()) # Output: Rabbit eats: clover

print("Octothorpe eats:", octothorpe.eats()) # Output: Octothorpe eats: campers

```

In this code, we define three classes `Bear`, `Rabbit`, and `Octothorpe`, each with the `eats()` method returning the respective food. We then create objects (`bear`, `rabbit`, and `octothorpe`) from each class and use the `eats()` method to print what each of them 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**.

Let's define the `Laser`, `Claw`, and `SmartPhone` classes, each with the `does()` method returning `'disintegrate'`, `'crush'`, and `'ring'`, respectively. Then, we'll define the `Robot` class, which will have one instance (object) of each of these classes. Finally, we'll define a `does()` method for the `Robot` class that will print what its component objects do:

```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("Laser:", self.laser.does())

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

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

# Create the Robot object

robot = Robot()

# Call the does() method of the Robot to print what its component objects do

robot.does()

```

Output:

```

Laser: disintegrate

Claw: crush

SmartPhone: ring

```

In this code, we define the `Laser`, `Claw`, and `SmartPhone` classes, each with the `does()` method returning the respective actions. Then, we define the `Robot` class with instances of each of these classes as its attributes. The `does()` method of the `Robot` class prints the actions performed by its component objects: Laser, Claw, and SmartPhone.