Name :- Deepawali . B. Mhaisagar
Assignment no 19

**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

Sure! Here's an example of creating a class called `Thing`, creating an object called `example` from that class, and printing both the class and the object:

```
class Thing:

    pass



# Create an instance of the Thing class

example = Thing()



# Print the class

print(Thing)



# Print the object

print(example)
```

When you run this code, the printed values for the class and the object will be different.

The output will be something like:

```
<class '__main__.Thing'>

<__main__.Thing object at 0x00000123ABCD>
```

The printed value for the class (`<class '__main__.Thing'>`) represents the class itself, indicating that it is an instance of the `Thing` class.

The printed value for the object (`<__main__.Thing object at 0x00000123ABCD>`) represents the instance of the `Thing` class (`example`), including the memory address where the object is stored. The memory address (`0x00000123ABCD`) will be different in your output.

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

2. Create a new class called Thing2 and add the value &#39;abc&#39; to the letters class attribute. Letters should be printed.

ANS

Certainly! Here's an example of creating a class called `Thing2`, adding the value `'abc'` to the `letters` class attribute, and printing the value:

```
class Thing2:

    letters = 'abc'



# Print the value of the letters class attribute

print(Thing2.letters)
```

When you run this code, it will print:

```
abc
```

In this example, we defined a class called `Thing2` and added the class attribute `letters` with the value `'abc'`. By accessing the class attribute using `Thing2.letters`, we can print its value, which is `'abc'` in this case.

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

In order to assign the value `'xyz'` to an instance attribute called `letters` in a class called `Thing3` and print its value, you will need to create an object (instance) of the class. Here's an example:

```
class Thing3:

    def __init__(self):

        self.letters = 'xyz'



# Create an instance of the Thing3 class

example = Thing3()



# Print the value of the letters instance attribute

print(example.letters)
```

When you run this code, it will print:

```
xyz
```

In this example, we defined a class called `Thing3` and added an instance attribute `letters` using the `__init__` method. The `__init__` method is a special method in Python classes that is executed when a new instance of the class is created. By assigning the value `'xyz'` to `self.letters` within the `__init__` method, we set the instance attribute `letters` to `'xyz'` for each object created from the class.

To access the instance attribute, we need to create an object (`example`) of the `Thing3` class. Then, we can print the value of the `letters` instance attribute using `example.letters`.

So, in this case, we need to create an object from the class in order to access and print the `letters` instance attribute.

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 creating an `Element` class with the instance attributes `name`, `symbol`, and `number`, and creating 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 an object of the Element class with the given values

element_object = Element('Hydrogen', 'H', 1)
```

In this example, we defined a class called `Element` with the `__init__` method. The `__init__` method is executed when a new object of the class is created. It takes the arguments `name`, `symbol`, and `number` to initialize the instance attributes `name`, `symbol`, and `number` respectively.

To create a class object with the given values, we instantiate the `Element` class by calling `Element('Hydrogen', 'H', 1)`, which creates an object with the name `'Hydrogen'`, symbol `'H'`, and number `1`. We assign this object to the variable `element_object`.

Now, you can access the instance attributes of the `element_object` object, for example:

```
print(element_object.name)    # Output: Hydrogen

print(element_object.symbol)  # Output: H

print(element_object.number)  # Output: 1
```

By accessing the instance attributes using dot notation (`element_object.name`, `element_object.symbol`, `element_object.number`), you can print their respective values.

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

To create a dictionary with the keys and values `'name': 'Hydrogen'`, `'symbol': 'H'`, and `'number': 1`, and then create an object called `hydrogen` from the `Element` class using this dictionary, you can follow these steps:

Step 1: Define the `Element` class with the `__init__` method:

```
class Element:
    def __init__(self, name, symbol, number):
        self.name = name
        self.symbol = symbol
        self.number = number
```

Step 2: Create the dictionary with the desired keys and values:

```
element_dict = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1}
```

Step 3: Create an object of the `Element` class using the dictionary values:

```
hydrogen = Element(**element_dict)
```

In this example, we created a dictionary called `element_dict` with the specified keys and values. The `**` notation in `Element(**element_dict)` is used to pass the dictionary items as keyword arguments to the `Element` class constructor (`__init__` method). This allows us to create an object (`hydrogen`) of the `Element` class with the values from the dictionary.

Now, you can access the instance attributes of the `hydrogen` object, for example:

```
print(hydrogen.name)    # Output: Hydrogen
print(hydrogen.symbol)  # Output: H
print(hydrogen.number)  # Output: 1
```

By accessing the instance attributes using dot notation (`hydrogen.name`, `hydrogen.symbol`, `hydrogen.number`), you can print their respective values. The values will match the ones specified in the 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.

ANS

To define a method called `dump()` in the `Element` class that prints the values of the object's attributes (`name`, `symbol`, and `number`), and then create the `hydrogen` object using this updated definition and use `dump()` to print its attributes, you can follow these steps:

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

# Use the dump() method to print the attributes
hydrogen.dump()
```

In this updated code, we added a method called `dump()` to the `Element` class. This method takes the `self` parameter, which represents the instance of the class. Inside the `dump()` method, we use `self.name`, `self.symbol`, and `self.number` to access the object's attributes and print their values.

When you run this code, it will output:

```
Name: Hydrogen
```

```
Symbol: H

Number: 1
```

By calling the `dump()` method on the `hydrogen` object, it will print the attributes' values of the object using the defined method.

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

When you call `print(hydrogen)`, it will first look for a special method called `__str__()` in the class definition. If `__str__()` is not defined, it falls back to the default string representation of the object, which includes the class name and memory address.

To update the class definition of `Element`, change the name of the method `dump()` to `__str__()` and define how the object should be printed as a string representation. Here's the modified 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}, Symbol: {self.symbol}, Number: {self.number}"



# Create the hydrogen object

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



# Call print(hydrogen)
```

```
print(hydrogen)
```

In this updated code, the `__str__()` method is defined in the `Element` class to customize the string representation of the object. Inside the `__str__()` method, we use string formatting to create a formatted string representation of the object's attributes.

When you call `print(hydrogen)`, it will invoke the `__str__()` method defined in the class and print the string representation returned by the method.

The output will be:

```
Name: Hydrogen, Symbol: H, Number: 1
```

By defining the `__str__()` method in the `Element` class, you can control how the object is represented as a string when using the `print()` function.

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

ANS

To modify the `Element` class to make the attributes `name`, `symbol`, and `number` private and define getter properties for each attribute to return their values, you can use Python's property decorators. Here's the updated code:

```
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 code, the attributes `name`, `symbol`, and `number` are prefixed with an underscore (`_`), indicating that they are intended to be private. We define getter properties using the `@property` decorator for each attribute. The getter properties allow us to access the values of the private attributes.

Now, you can create an `Element` object and access the attribute values using the getter properties:

```
# Create the hydrogen object
hydrogen = Element('Hydrogen', 'H', 1)


# Access the attribute values using the getter properties
print(hydrogen.name)    # Output: Hydrogen
print(hydrogen.symbol)  # Output: H
print(hydrogen.number)  # Output: 1
```

By accessing `hydrogen.name`, `hydrogen.symbol`, and `hydrogen.number`, you can retrieve the values of the private attributes `name`, `symbol`, and `number` respectively. The getter properties provide a controlled way to access the private attribute values.

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

Sure! Here's an example of defining three classes: `Bear`, `Rabbit`, and `Octothorpe`, each with a single method `eats()` that returns a specific food item. 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 one object from each class
bear = Bear()
rabbit = Rabbit()
octothorpe = Octothorpe()

# Print what each object eats
print(bear.eats())       # Output: berries
print(rabbit.eats())     # Output: clover
print(octothorpe.eats())  # Output: campers
```

In this code, we define three classes: `Bear`, `Rabbit`, and `Octothorpe`, each with a single method `eats()` that returns a specific food item for that particular animal. We then create one object from each class (`bear`, `rabbit`, and `octothorpe`), and use the `eats()` method to print what each object eats.

The output will be:

```
berries
clover
campers
```

By calling the `eats()` method on each object, we retrieve the specific food item associated with that class.

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

Certainly! Here's an example of defining the `Laser`, `Claw`, and `SmartPhone` classes, each with a single method `does()` that returns a specific action. Then, we define the `Robot` class that has one instance (object) of each of these classes, and define a `does()` method for the `Robot` class 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 of the Robot
robot.does()
```

In this code, we define three classes: `Laser`, `Claw`, and `SmartPhone`, each with a single method `does()` that returns a specific action. Then, we define the `Robot` class with an `__init__` method that creates instances (objects) of each of these classes as its components. The `Robot` class also has a `does()` method that calls the `does()` method of each component object and prints what they do.

When you run this code, it will output:

```
Laser: disintegrate
Claw: crush
SmartPhone: ring
```

By calling the `does()` method of the `Robot` object, it will print what each of its component objects does. The output shows the actions associated with each component: `disintegrate` for the `Laser`, `crush` for the `Claw`, and `ring` for the `SmartPhone`.