Q1. What is the difference between \_\_getattr\_\_ and \_\_getattribute\_\_?

Q2. What is the difference between properties and descriptors?

Q3. What are the key differences in functionality between \_\_getattr\_\_ and \_\_getattribute\_\_, as well as

properties and descriptors?

### **Q1. What is the difference between \_\_getattr\_\_ and \_\_getattribute\_\_?**

**\_\_getattribute\_\_**:

* **Purpose**: This method is called automatically when any attribute of an object is accessed, whether the attribute exists or not.
* **Usage**: It allows you to define custom behavior for all attribute access on the object.
* **Signature**: \_\_getattribute\_\_(self, name)

**Example**:  
python  
Copy code  
class MyClass:

def \_\_getattribute\_\_(self, name):

print(f"Getting attribute: {name}")

return super().\_\_getattribute\_\_(name)

obj = MyClass()

obj.some\_attr # Outputs: Getting attribute: some\_attr

**\_\_getattr\_\_**:

* **Purpose**: This method is called only when accessing an attribute that does not exist on the object. It acts as a fallback if the attribute is not found by \_\_getattribute\_\_.
* **Usage**: It is used to define custom behavior for attributes that are not present in the instance’s \_\_dict\_\_.
* **Signature**: \_\_getattr\_\_(self, name)

**Example**:  
python  
Copy code  
class MyClass:

def \_\_getattr\_\_(self, name):

print(f"Attribute {name} not found")

return "default value"

obj = MyClass()

print(obj.some\_attr) # Outputs: Attribute some\_attr not found and "default value"

**Key Differences**:

* \_\_getattribute\_\_ is always called for every attribute access, while \_\_getattr\_\_ is only called when the attribute is not found.
* \_\_getattribute\_\_ can be used for more general attribute handling, including existing attributes, whereas \_\_getattr\_\_ is specifically for handling missing attributes.

### **Q2. What is the difference between properties and descriptors?**

**Properties**:

* **Purpose**: Properties are a built-in Python feature that allows you to define methods in a class that can be accessed like attributes. They provide a way to manage attribute access and encapsulate logic without changing the interface of the class.
* **Implementation**: Defined using the property decorator or function.

**Example**:  
python  
Copy code  
class MyClass:

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

self.\_value = value

@property

def value(self):

return self.\_value

@value.setter

def value(self, new\_value):

self.\_value = new\_value

obj = MyClass(10)

print(obj.value) # Outputs: 10

obj.value = 20

print(obj.value) # Outputs: 20

**Descriptors**:

* **Purpose**: Descriptors are a more general mechanism for attribute access control in Python. They are used to manage how attributes are accessed and modified. A descriptor is any object that implements at least one of the descriptor methods (\_\_get\_\_, \_\_set\_\_, \_\_delete\_\_).
* **Implementation**: Defined by creating a class that implements one or more descriptor methods.

**Example**:  
python  
Copy code  
class Descriptor:

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

self.value = value

def \_\_get\_\_(self, instance, owner):

return self.value

def \_\_set\_\_(self, instance, value):

self.value = value

class MyClass:

attr = Descriptor(10)

obj = MyClass()

print(obj.attr) # Outputs: 10

obj.attr = 20

print(obj.attr) # Outputs: 20

**Key Differences**:

* **Properties**: Simple and convenient way to manage attribute access using decorators. Suitable for most use cases where you want to encapsulate logic for getting and setting attributes.
* **Descriptors**: More flexible and powerful, allowing you to manage attribute access across multiple classes. Descriptors are useful when you need to apply the same attribute management logic across different classes or need more control over attribute access.

### **Q3. What are the key differences in functionality between \_\_getattr\_\_ and \_\_getattribute\_\_, as well as properties and descriptors?**

**Functionality Differences**:

* **\_\_getattr\_\_ vs. \_\_getattribute\_\_**:
  + \_\_getattribute\_\_ is invoked for every attribute access, regardless of whether the attribute exists or not. It provides a way to intercept all attribute access.
  + \_\_getattr\_\_ is invoked only when an attribute is not found in the instance's dictionary or its class hierarchy. It provides a fallback mechanism for missing attributes.
* **Properties vs. Descriptors**:
  + **Properties**: Simplified attribute access control using decorators. Useful for adding getter, setter, and deleter methods to manage attributes in a straightforward way.
  + **Descriptors**: More versatile and can be used across multiple classes. They involve implementing methods to manage attribute access and modification directly. Descriptors offer a higher level of control and are suitable for complex attribute management scenarios.

**Summary**:

* Use \_\_getattribute\_\_ for general-purpose attribute access handling and \_\_getattr\_\_ for handling missing attributes specifically.
* Use properties for simple attribute management and encapsulation within a single class, and descriptors for more complex or cross-class attribute management scenarios.