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

ANS: Both `\_\_getattr\_\_` and `\_\_getattribute\_\_` are special methods in Python classes that deal with attribute access. However, they have different purposes and behaviors:

A. \*\*`\_\_getattr\_\_(self, name)`\*\*:

- This method is called when an attribute lookup (both read and write) fails in the usual way.

- It is a fallback method that is invoked only when the requested attribute is not found via the usual lookup process.

- The method takes two arguments: `self` (the instance of the class) and `name` (the name of the attribute being accessed).

- It should return the value of the attribute or raise an `AttributeError` if the attribute cannot be found.

Example of `\_\_getattr\_\_`:

class MyClass:

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

print(f'Attribute {name} not found!')

return None

obj = MyClass()

print(obj.some\_attribute) # Output: Attribute some\_attribute not found! None

B. \*\*`\_\_getattribute\_\_(self, name)`\*\*:

- This method is called for every attribute access, whether the attribute exists or not.

- It is called before the usual attribute lookup process and allows you to intercept and customize the attribute access behavior for all attributes.

- The method takes two arguments: `self` (the instance of the class) and `name` (the name of the attribute being accessed).

- It should return the value of the attribute.

- Be cautious when using `\_\_getattribute\_\_` as it can lead to infinite recursion if not implemented carefully. If you need to access the attribute within `\_\_getattribute\_\_`, it is recommended to use `super().\_\_getattribute\_\_(name)`.

Example of `\_\_getattribute\_\_`:

```python

class MyClass:

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

self.some\_attribute = 42

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

print(f'Getting attribute {name}')

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

obj = MyClass()

print(obj.some\_attribute) # Output: Getting attribute some\_attribute 42

Q2. What is the difference between properties and descriptors?

ANS: Properties and descriptors are both mechanisms in Python that allow you to control access to attributes of an object, but they serve different purposes and have different levels of flexibility and complexity.

\*\*Properties:\*\*

- Properties are a simple and convenient way to add getter, setter, and deleter methods to an attribute in a class.

- They are defined using the `@property` decorator for the getter method, `@<attribute\_name>.setter` decorator for the setter method, and `@<attribute\_name>.deleter` decorator for the deleter method.

- Properties are limited to controlling access to a single attribute of the class.

- They are used to provide a more controlled interface to the class's attributes and are often used to ensure data validation or computation upon attribute access.

- Properties are bound to the attribute name and rely on Python's attribute lookup mechanism.

Example of using properties:

class Circle:

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

self.\_radius = radius

@property

def radius(self):

return self.\_radius

@radius.setter

def radius(self, value):

if value <= 0:

raise ValueError("Radius must be positive.")

self.\_radius = value

@radius.deleter

def radius(self):

del self.\_radius

\*\*Descriptors:\*\*

- Descriptors are more powerful and flexible than properties. They allow you to define custom classes to control attribute access at a lower level.

- A descriptor is an object that implements any combination of `\_\_get\_\_`, `\_\_set\_\_`, and `\_\_delete\_\_` methods. These methods allow you to define the behavior when getting, setting, or deleting an attribute.

- Descriptors can be reused for multiple attributes in a class or even across different classes, providing a more modular and reusable approach to controlling attribute access.

- Descriptors are typically used when you want to enforce common behavior across different attributes or perform more complex operations during attribute access.

Example of using a descriptor:

class PositiveValue:

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

return instance.\_value

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

if value <= 0:

raise ValueError("Value must be positive.")

instance.\_value = value

class Circle:

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

self.\_radius = PositiveValue()

self.radius = radius

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

ANS: \*\*`\_\_getattr\_\_` vs. `\_\_getattribute\_\_`:\*\*

I. \*\*Invocation:\*\*

- `\_\_getattr\_\_(self, name)` is invoked when an attribute lookup fails. It is only called if the attribute is not found through the regular attribute lookup process.

- `\_\_getattribute\_\_(self, name)` is called for every attribute access, regardless of whether the attribute exists or not. It is called before the usual attribute lookup process.

II. \*\*Attribute Access:\*\*

- `\_\_getattr\_\_` is a fallback method used to handle non-existing attributes.

- `\_\_getattribute\_\_` is a method that is called for every attribute access, even if the attribute exists.

III. \*\*Return Value:\*\*

- `\_\_getattr\_\_` should return the value of the attribute or raise an `AttributeError` if the attribute cannot be found.

- `\_\_getattribute\_\_` should return the value of the attribute.

\*\*Properties vs. Descriptors:\*\*

I. \*\*Level of Control:\*\*

- Properties are a high-level way to control access to a single attribute. They provide a simple way to add getter, setter, and deleter methods to an attribute in a class.

- Descriptors offer a lower-level and more flexible mechanism for controlling attribute access. They allow you to define custom classes that control how attributes are accessed, set, or deleted at a lower level.

II. \*\*Scope of Control:\*\*

- Properties are specific to individual attributes in a class. You need to define a property for each attribute you want to control.

- Descriptors can be reused for multiple attributes in a class or even across different classes. They provide a more modular and reusable approach to controlling attribute access.

III. \*\*Implementation:\*\*

- Properties are implemented using the `@property`, `@<attribute\_name>.setter`, and `@<attribute\_name>.deleter` decorators.

- Descriptors are implemented by creating a separate class that defines `\_\_get\_\_`, `\_\_set\_\_`, and/or `\_\_delete\_\_` methods, which are then used as descriptors for specific attributes in the class.

IV. \*\*Control Over Get, Set, and Delete:\*\*

- With properties, you can control the behavior of attribute getting, setting, and deleting at a higher level using the corresponding methods.

- Descriptors give you fine-grained control over how attributes are accessed, set, or deleted, allowing you to perform complex operations during these actions.