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

The main difference between `\_\_getattr\_\_` and `\_\_getattribute\_\_` in Python lies in how they handle attribute access for objects.

1. `\_\_getattr\_\_`:

- `\_\_getattr\_\_` is a special method that gets called when an attribute lookup fails. It is only invoked if the requested attribute is not found through the normal lookup process.

- This method is useful for implementing "attribute not found" behavior or dynamically providing attributes. It receives the name of the attribute as a parameter and can return a value or raise an `AttributeError` if desired.

- Note that `\_\_getattr\_\_` is only called for attributes that are not found through the usual attribute lookup, meaning it won't be triggered for existing attributes or methods.

Example:

```python

class Example:

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

# Handle attribute not found

return f'Attribute {name} not found.'

obj = Example()

print(obj.some\_attribute) # Output: Attribute some\_attribute not found.

```

2. `\_\_getattribute\_\_`:

- `\_\_getattribute\_\_` is a special method that gets called for every attribute access, regardless of whether the attribute exists or not.

- This method allows you to override the default attribute access behavior completely. It is called before `\_\_getattr\_\_` and other attribute lookup mechanisms.

- Caution should be exercised when implementing `\_\_getattribute\_\_` as it can lead to infinite recursion if not handled properly. If you need to access attributes within `\_\_getattribute\_\_`, you should use `super().\_\_getattribute\_\_(name)` to avoid recursion.

Example:

```python

class Example:

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

# Custom attribute access behavior

return f'Accessing attribute {name}.'

obj = Example()

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

```

In summary, `\_\_getattr\_\_` is called when an attribute is not found, while `\_\_getattribute\_\_` is called for every attribute access, regardless of whether the attribute exists or not. You can use these methods to customize attribute access behavior and handle attribute lookup dynamically in Python objects.

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

Properties and descriptors are both mechanisms in Python that allow you to control attribute access and provide customized behavior. While they serve similar purposes, there are some differences between them:

Properties:

- Properties are a high-level and convenient way to manage attribute access in Python.

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

- Properties allow you to define methods that are accessed like attributes, providing a way to encapsulate attribute logic and apply getters, setters, and deleters transparently.

- Properties are defined at the class level and associated with a specific attribute name.

- Properties are typically used when you want to add additional behavior or computation to attribute access without requiring changes to the client code.

Example of using properties:

```python

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

circle = Circle(5)

print(circle.radius) # Output: 5

circle.radius = 10

print(circle.radius) # Output: 10

```

Descriptors:

- Descriptors are a lower-level mechanism that allows you to define custom attribute access behavior at the class level.

- They involve implementing the `\_\_get\_\_`, `\_\_set\_\_`, and `\_\_delete\_\_` methods as part of a descriptor class.

- Descriptors provide fine-grained control over attribute access, allowing you to define custom logic for getting, setting, and deleting attributes.

- Descriptors are typically used when you need to apply the same behavior to multiple attributes or when you want to control attribute access on a class level.

Example of using descriptors:

```python

class Descriptor:

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

# Custom logic for getting attribute

# `instance` is the instance of the class

# `owner` is the class itself

return instance.\_value

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

# Custom logic for setting attribute

instance.\_value = value

class MyClass:

attribute = Descriptor()

my\_obj = MyClass()

my\_obj.attribute = 42

print(my\_obj.attribute) # Output: 42

```

In summary, properties provide a higher-level interface for managing attribute access and allow you to define custom behavior using decorators. Descriptors, on the other hand, provide lower-level control over attribute access by implementing the descriptor protocol. They are typically used when you need more granular control or when you want to apply the same behavior to multiple attributes in a class.

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

The key differences in functionality between `\_\_getattr\_\_`, `\_\_getattribute\_\_`, properties, and descriptors are as follows:

1. `\_\_getattr\_\_` vs. `\_\_getattribute\_\_`:

- `\_\_getattr\_\_` is called only when an attribute is not found through the normal attribute lookup process. It allows you to define custom behavior for handling attribute access when the attribute is not present.

- `\_\_getattribute\_\_` is called for every attribute access, regardless of whether the attribute exists or not. It allows you to override the default attribute access behavior completely.

- `\_\_getattribute\_\_` is typically used when you want to intercept and customize all attribute access, while `\_\_getattr\_\_` is used when you specifically want to handle attribute access for non-existing attributes.

2. Properties vs. Descriptors:

- Properties provide a high-level and convenient way to manage attribute access. They allow you to define methods that are accessed like attributes, providing transparent getter, setter, and deleter behavior.

- Descriptors provide a lower-level mechanism for controlling attribute access at the class level. They involve implementing the `\_\_get\_\_`, `\_\_set\_\_`, and `\_\_delete\_\_` methods as part of a descriptor class.

- Properties are typically used to add additional behavior or computation to attribute access without requiring changes to the client code. They are defined at the class level and associated with a specific attribute name.

- Descriptors are used when you need fine-grained control over attribute access, want to apply the same behavior to multiple attributes, or want to control attribute access on a class level. Descriptors are typically defined separately from the attributes they manage and can be reused across multiple classes.

In summary, `\_\_getattr\_\_` and `\_\_getattribute\_\_` control attribute access at the object level, with `\_\_getattr\_\_` handling non-existing attributes and `\_\_getattribute\_\_` handling all attributes. Properties provide a high-level interface for managing attribute access with decorators, while descriptors offer a lower-level mechanism for customizing attribute access at the class level.