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

A. **\_getattr\_\_** and **\_\_getattribute\_\_** are both methods in Python that are used for attribute access, but they serve slightly different purposes:

1. **\_\_getattr\_\_(self, name)**: This method is called when an attempt is made to access an attribute that doesn't exist in the instance. It takes two arguments: **self** (the instance) and **name** (the name of the attribute being accessed). If **\_\_getattr\_\_** is defined in a class, it is only invoked for attributes that are not found through the normal lookup process.

Example:

class Example:

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

return f"Attribute {name} not found!"

obj = Example()

print(obj.undefined\_attr) # Output: Attribute undefined\_attr not found!

class Example:

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

return f"Attribute {name} not found!"

obj = Example()

print(obj.undefined\_attr) # Output: Attribute undefined\_attr not found!

class Example:

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

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

# To avoid infinite recursion, call the parent class's \_\_getattribute\_\_

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

obj = Example()

obj.some\_attr # Output: Accessing attribute: some\_attr

`\_\_getattr\_\_` and `\_\_getattribute\_\_` are both methods in Python that are used for attribute access, but they serve slightly different purposes:

1. `\_\_getattr\_\_(self, name)`: This method is called when an attempt is made to access an attribute that doesn't exist in the instance. It takes two arguments: `self` (the instance) and `name` (the name of the attribute being accessed). If `\_\_getattr\_\_` is defined in a class, it is only invoked for attributes that are not found through the normal lookup process.

Example:

```python

class Example:

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

return f"Attribute {name} not found!"

obj = Example()

print(obj.undefined\_attr) # Output: Attribute undefined\_attr not found!

```

2. `\_\_getattribute\_\_(self, name)`: This method is called every time an attribute is accessed on an instance, regardless of whether the attribute exists or not. It is a more general mechanism compared to `\_\_getattr\_\_` because it's invoked for every attribute access. This method can be used to implement custom behavior for attribute access, such as logging, validation, or dynamic attribute lookup.

Example:

```python

class Example:

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

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

# To avoid infinite recursion, call the parent class's \_\_getattribute\_\_

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

obj = Example()

obj.some\_attr # Output: Accessing attribute: some\_attr

```

In summary, `\_\_getattr\_\_` is called only when an attribute is not found through the normal lookup process, while `\_\_getattribute\_\_` is called for every attribute access, allowing for more fine-grained control over attribute access behavior. However, it's important to use `\_\_getattribute\_\_` with caution, as incorrectly implemented can lead to infinite recursion.

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

A. Properties and descriptors are closely related concepts in Python, particularly within the context of attribute access and manipulation. Here's a breakdown of their differences:

1. **Properties**:
   * Properties are a high-level way of defining getter, setter, and deleter methods for a class attribute.
   * They are created using the **property()** function or by using the **@property** decorator.
   * Properties allow you to define custom behavior for getting, setting, and deleting attribute values, while still accessing them like normal attributes.
   * They are often used to implement computed attributes, where the value of the attribute is calculated dynamically based on other attributes.
   * Properties provide a clean syntax for attribute access and can hide the complexity of attribute manipulation.

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

c = Circle(5)

print(c.radius) # Output: 5

c.radius = 10

print(c.radius) # Output: 10

1. **Descriptors**:
   * Descriptors are a lower-level protocol for defining how attribute access is handled in Python.
   * They are defined by implementing one or more of the **\_\_get\_\_**, **\_\_set\_\_**, and **\_\_delete\_\_** methods.
   * Descriptors are used to customize attribute access at the class level, rather than the instance level.
   * They can be reused across multiple classes to provide consistent behavior for different attributes.
   * Descriptors are more powerful and flexible than properties but also more complex to implement and use directly.

class NonNegative:

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

self.name = name

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

return instance.\_\_dict\_\_[self.name]

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

if value < 0:

raise ValueError("Value cannot be negative")

instance.\_\_dict\_\_[self.name] = value

class Circle:

radius = NonNegative('radius')

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

self.radius = radius

c = Circle(5)

print(c.radius) # Output: 5

c.radius = 10

print(c.radius) # Output: 10Properties and descriptors are closely related concepts in Python, particularly within the context of attribute access and manipulation. Here's a breakdown of their differences:

1. \*\*Properties\*\*:

- Properties are a high-level way of defining getter, setter, and deleter methods for a class attribute.

- They are created using the `property()` function or by using the `@property` decorator.

- Properties allow you to define custom behavior for getting, setting, and deleting attribute values, while still accessing them like normal attributes.

- They are often used to implement computed attributes, where the value of the attribute is calculated dynamically based on other attributes.

- Properties provide a clean syntax for attribute access and can hide the complexity of attribute manipulation.

Example:

```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

c = Circle(5)

print(c.radius) # Output: 5

c.radius = 10

print(c.radius) # Output: 10

```

2. \*\*Descriptors\*\*:

- Descriptors are a lower-level protocol for defining how attribute access is handled in Python.

- They are defined by implementing one or more of the `\_\_get\_\_`, `\_\_set\_\_`, and `\_\_delete\_\_` methods.

- Descriptors are used to customize attribute access at the class level, rather than the instance level.

- They can be reused across multiple classes to provide consistent behavior for different attributes.

- Descriptors are more powerful and flexible than properties but also more complex to implement and use directly.

Example:

```python

class NonNegative:

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

self.name = name

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

return instance.\_\_dict\_\_[self.name]

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

if value < 0:

raise ValueError("Value cannot be negative")

instance.\_\_dict\_\_[self.name] = value

class Circle:

radius = NonNegative('radius')

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

self.radius = radius

c = Circle(5)

print(c.radius) # Output: 5

c.radius = 10

print(c.radius) # Output: 10

```

In summary, properties provide a simpler and more concise way of achieving similar functionality to descriptors, but descriptors offer more control and flexibility over attribute access behavior.

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

A. Both `\_\_getattr\_\_` and `\_\_getattribute\_\_` are methods in Python classes that are related to attribute access, but they serve different purposes:

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

- This method is called when the requested attribute is not found through the normal lookup process.

- It allows you to define custom behavior for accessing attributes that don't exist.

- It's only invoked when an attribute is not found through the usual means, such as when using dot notation (`obj.attr`).

- It's commonly used for implementing dynamic attributes or handling attribute delegation.

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

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

- It allows you to intercept all attribute accesses and customize the behavior.

- You need to be cautious when implementing `\_\_getattribute\_\_` because calling other methods or accessing attributes within it can lead to infinite recursion if not handled properly.

Regarding properties and descriptors:

1. Properties:

- Properties allow you to define custom behavior for getting, setting, and deleting attributes.

- They are created using the `property` built-in function or the `@property` decorator.

- Properties are typically used to manage attribute access for a single attribute of an object.

- They provide a convenient way to encapsulate attribute access logic.

2. Descriptors:

- Descriptors are a more general mechanism for defining custom behavior for attribute access.

- They are implemented by defining classes with `\_\_get\_\_`, `\_\_set\_\_`, and `\_\_delete\_\_` methods.

- Descriptors can be used to customize attribute access for multiple attributes of a class or even multiple classes.

- They offer more flexibility and reusability compared to properties but may require more complex implementation.

In summary, `\_\_getattr\_\_` and `\_\_getattribute\_\_` are methods for customizing attribute access at the instance level, while properties and descriptors provide mechanisms for defining custom attribute access behavior at the class level. Properties are a simpler way to achieve attribute customization for a single attribute, while descriptors offer more flexibility and reusability for more complex scenarios.