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

\_\_getattr\_\_:

Called when an attribute lookup has not found the attribute in the usual places (i.e. it is not an instance attribute nor is it found in the class tree for self). name is the attribute name. This method should return the (computed) attribute value or raise an AttributeError exception.

Note that if the attribute is found through the normal mechanism, \_\_getattr\_\_() is not called. (This is an intentional asymmetry between \_\_getattr\_\_() and \_\_setattr\_\_().) This is done both for efficiency reasons and because otherwise \_\_getattr\_\_() would have no way to access other attributes of the instance. Note that at least for instance variables, you can fake total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the \_\_getattribute\_\_() method below for a way to actually get total control in new-style classes.

\_\_getattribute\_\_:

Called unconditionally to implement attribute accesses for instances of the class. If the class also defines \_\_getattr\_\_(), the latter will not be called unless \_\_getattribute\_\_() either calls it explicitly or raises an AttributeError. This method should return the (computed) attribute value or raise an AttributeError exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, object.\_\_getattribute\_\_(self, name).

Code:

class Yeah(object):

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

self.name = name

# Gets called when an attribute is accessed

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

print '\_\_getattribute\_\_ ', item

# Calling the super class to avoid recursion

return super(Yeah, self).\_\_getattribute\_\_(item)

# Gets called when the item is not found via \_\_getattribute\_\_

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

print '\_\_getattr\_\_ ', item

return super(Yeah, self).\_\_setattr\_\_(item, 'orphan')

>> y1 = Yeah('yes')

>> y1.name

\_\_getattribute\_\_ name

'yes'

>> y1.foo

\_\_getattribute\_\_ foo

\_\_getattr\_\_ foo

>> y1.foo

\_\_getattribute\_\_ foo

'orphan'

>> y1.goo

\_\_getattribute\_\_ goo

\_\_getattr\_\_ goo

>> y1.\_\_dict\_\_

\_\_getattribute\_\_ \_\_dict\_\_

{'\_\_members\_\_': 'orphan',

'\_\_methods\_\_': 'orphan',

'foo': 'orphan',

'goo': 'orphan',

'name': 'yes'}

Q2. What is the difference between properties and descriptors?

**Descriptors:**

Descriptors are a specific Python feature that power a lot of the magic hidden under the language’s hood. Descriptors are Python objects that implement a method of the descriptor protocol, which gives you the ability to create objects that have special behavior when they’re accessed as attributes of other objects. Here you can see the correct definition of the descriptor protocol:

\_\_get\_\_(self, obj, type=None) -> object

\_\_set\_\_(self, obj, value) -> None

\_\_delete\_\_(self, obj) -> None

\_\_set\_name\_\_(self, owner, name)

If your descriptor implements just .\_\_get\_\_(), then it’s said to be a non-data descriptor. If it implements .\_\_set\_\_() or .\_\_delete\_\_(), then it’s said to be a data descriptor. Note that this difference is not just about the name, but it’s also a difference in behavior. That’s because data descriptors have precedence during the lookup process

Code:

# descriptors.py

class Verbose\_attribute():

def \_\_get\_\_(self, obj, type=None) -> object:

print("accessing the attribute to get the value")

return 42

def \_\_set\_\_(self, obj, value) -> None:

print("accessing the attribute to set the value")

raise AttributeError("Cannot change the value")

class Foo():

attribute1 = Verbose\_attribute()

my\_foo\_object = Foo()

x = my\_foo\_object.attribute1

print(x)

In the example above, Verbose\_attribute() implements the descriptor protocol. Once it’s instantiated as an attribute of Foo, it can be considered a descriptor.

As a descriptor, it has binding behavior when it’s accessed using dot notation. In this case, the descriptor logs a message on the console every time it’s accessed to get or set a value:

* When it’s accessed to .\_\_get\_\_() the value, it always returns the value 42.
* When it’s accessed to .\_\_set\_\_() a specific value, it raises an AttributeError exception, which is the recommended way to implement read-only descriptors.

In the example above and you’ll see the descriptor log the access to the console before returning the constant value. When you try to access attribute1, the descriptor logs this access to the console, as defined in .\_\_get\_\_().

**Property:**

If you want to get the same result as the previous example without explicitly using a Python descriptor, then the most straightforward approach is to use a property. The following example uses a property that logs a message to the console when it’s accessed:

Code:

# property\_decorator.py

class Foo():

@property

def attribute1(self) -> object:

print("accessing the attribute to get the value")

return 42

@attribute1.setter

def attribute1(self, value) -> None:

print("accessing the attribute to set the value")

raise AttributeError("Cannot change the value")

my\_foo\_object = Foo()

x = my\_foo\_object.attribute1

print(x)

The example above makes use of decorators to define a property with attached getter and setter methods. But as you may know, decorators are just syntactic sugar. The example before, in fact, can be written as follows:

Code:

# property\_function.py

class Foo():

def getter(self) -> object:

print("accessing the attribute to get the value")

return 42

def setter(self, value) -> None:

print("accessing the attribute to set the value")

raise AttributeError("Cannot change the value")

attribute1 = property(getter, setter)

my\_foo\_object = Foo()

x = my\_foo\_object.attribute1

print(x)

Now you can see that the property has been created by using property(). The signature of this function is as follows:

property(fget=None, fset=None, fdel=None, doc=None) -> object

property() returns a property object that implements the descriptor protocol. It uses the parameters fget, fset and fdel for the actual implementation of the three methods of the protocol.

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

The main difference between \_\_getattr\_\_ and \_\_getattribute\_\_ is that if the attribute was not found by the usual way then \_\_getattr\_\_ is used. Whereas the \_\_getattribute\_\_ is used before looking at the actual attributes on the object.

Descriptors are a low-level mechanism that lets you hook into an object's attributes being accessed. Properties are a high-level application of this; that is, properties are implemented using descriptors.