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

\_\_getattribute\_\_ has a default implementation, but \_\_getattr\_\_ does not. This has a clear meaning: since \_\_getattribute\_\_ has a default implementation, while \_\_getattr\_\_ not, clearly python encourages users to implement \_\_getattr\_\_ .

* **getattribute**: Is used to retrieve an attribute from an instance. It captures every attempt to access an instance attribute by using dot notation or getattr() built-in function.
* **getattr**: Is executed as the last resource when attribute is not found in an object. You can choose to return a default value or to raise AttributeError.

Going back to the **\_\_getattribute\_\_** function; if the default implementation was not overridden; the following checks are done when executing the method:

* Check if there is a descriptor with the same name (attribute name) defined in any class in the MRO chain (method object resolution)
* Then looks into the instance’s namespace
* Then looks into the class namespace
* Then into each base’s namespace and so on.
* Finally, if not found, the default implementation calls the fallback **getattr**() method of the instance and it raises an AttributeError exception as default implementation.

This is the actual *implementation* of the object.\_\_getattribute\_\_ method:

In reading through Beazley & Jones PCB, I have stumbled on an explicit and practical use-case for \_\_getattr\_\_ that helps answer the "when" part of the OP's question. From the book:

"The \_\_getattr\_\_() method is kind of like a catch-all for attribute lookup. It's a method that gets called if code tries to access an attribute that doesn't exist." We know this from the above answers, but in PCB recipe 8.15, this functionality is used to implement the **delegation design pattern**. If Object A has an attribute Object B that implements many methods that Object A wants to delegate to, rather than redefining all of Object B's methods in Object A just to call Object B's methods, define a \_\_getattr\_\_() method as follows:

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

return getattr(self.\_b, name)

where \_b is the name of Object A's attribute that is an Object B. When a method defined on Object B is called on Object A, the \_\_getattr\_\_ method will be invoked at the end of the lookup chain. This would make code cleaner as well, since you do not have a list of methods defined just for delegating to another object.

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

The Cliff's Notes version: 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.

**Descriptor example:**

class Celsius( object ):

def \_\_init\_\_( self, value=0.0 ):

self.value= float(value)

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

return self.value

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

self.value= float(value)

class Farenheit( object ):

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

return instance.celsius \* 9 / 5 + 32

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

instance.celsius= (float(value)-32) \* 5 / 9

class Temperature( object ):

celsius= Celsius()

farenheit= Farenheit()

>>>

oven= Temperature()

>>>

oven.farenheit= 450

>>>

oven.celsius

232.22222222222223

>>>

oven.celsius= 175

>>>

oven.farenheit

347.0

**Property example:**

class Temperature( object ):

def fget( self ):

return self.celsius \* 9 / 5 + 32

def fset( self, value ):

self.celsius= (float(value)-32) \* 5 / 9

farenheit= property( fget, fset )

def cset( self, value ):

self.cTemp= float(value)

def cget( self ):

return self.cTemp

celsius= property( cget, cset, doc="Celsius temperature" )

>>>

oven= Temperature()

>>>

oven.farenheit= 450

>>>

oven.celsius

232.22222222222223

>>>

oven.celsius= 175

>>>

oven.farenheit

347.0

The property function gives us a handy way to implement a simple descriptor without defining a separate class. Rather than create a complete class definition, we can write getter and setter method functions, and then bind these functions to an attribute name.

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

So, the difference between \_\_getattribute\_\_() and \_\_getattr\_\_() is that the first one is called unconditionally when an attribute is being retrieved from an instance while the second is called only when the attribute was not found.

# The difference between \_\_getattribute\_\_ and \_\_getattr\_\_

The other day I was teaching Python meta-programming to a workmate. I think it’s a good way to learn about high order functions since meta-programming makes extensive use of closures, function builders, decorators… I was trying to make a concept probe about a very, very generic REST connector. Here is my first (and wrong) attempt:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class RESTConn(object):      def \_\_init\_\_(self, entry\_point):      self.entry\_point = entry\_point      def method\_builder(self, method\_name):      verb, \_, collection = method\_name.split('\_', 2)      def do\_verb(payload=None, \*\*kwargs):        uri = self.make\_uri(collection)        querystring = self.make\_querystring(kwargs)        print verb.upper(), self.combine(uri, querystring)        if payload:          print payload        return do\_verb      def make\_uri(self, collection):      return '/'.join([self.entry\_point, collection])      def make\_querystring(self, kwargs):      return '&'.join(['='.join(pair) for pair in kwargs.iteritems()])      def combine(self, uri, querystring):      if querystring:        return '&'.join([uri, querystring])        return uri      def \_\_getattribute\_\_(self, name):      if not hasattr(self, name):        method = self.method\_builder(name)        setattr(self, name, method)        return super(RESTConn, self).\_\_getattribute\_\_(name) |

c = RESTConn('unoyunodiez.com')

c.get\_from\_articles()

The program falls into an infinite recursion and do nothing before crashing. Why?

There are three problems here. First and most important is using \_\_getattribute\_\_(), second is using hasattr() and third is accessing self.method\_builder().

The object’s method \_\_getattribute\_\_() is used to retrieve an attribute from an instance. **It captures every attempt** to access an instance attribute by using dot notation or getattr() built-in function. Unless it was overridden, the former expression is translated into object.\_\_getattribute\_\_(self, ‘get\_from\_article’). The default implementations looks into the instance’s namespace, then looks into the class namespace, then into each base’s namespace and so on. Finally, if not found, the default implementation calls the fallback \_\_getattr\_\_() method of the instance and it raises an AttributeError exception as default implementation.

This is not a problem by itself but if you pay attention enough you’ll notice we are trying to create the new method only if the object does not have the method yet. It is semantically the same as overriding \_\_getattr\_\_() because **it is called only when the attribute was not found**. So, even if we cannot explain the infinite recursion error yet, we can fix the class by replacing:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def \_\_getattribute\_\_(self, name):    if not hasattr(self, name):      method = self.method\_builder(name)      setattr(self, name, method)      return super(RESTConn, self).\_\_getattribute\_\_(name) |

with:

|  |  |
| --- | --- |
| 1  2  3  4 | def \_\_getattr\_\_(self, name):    method = self.method\_builder(name)    setattr(self, name, method)    return getattr(self, name) |

So, the difference between \_\_getattribute\_\_() and \_\_getattr\_\_() is that the first one is called unconditionally when an attribute is being retrieved from an instance while the second is called only when the attribute was not found.

But, what about the infinite recursion? Why the first example was failing?

Ok, I said the problems were using \_\_getattribute\_\_, hasattr() and self.method\_builder() but the **real problem is using hasattr() and self.method\_builder() inside \_\_getattribute\_\_()**. According to Python documentation [hasattr(self, name) is implemented by trying getattr(self, name) and catching exceptions](http://docs.python.org/2/library/functions.html" \l "hasattr" \o "hasattr() documentation" \t "_blank). If an exception is caught, then hasattr() returns False. But using getattr() is an attempt to access an instance attribute so \_\_getattribute\_\_() is called again. This leads to another call to hasattr() which use getattr() which is another attempt to find and attribute and it calls \_\_getattribute\_\_() which call hasattr() which use getattr() and on and on and on… until a RuntimeError exception is raised. So all the hasattr() return False and the program can continue with the next statement.

The next statement is more dangerous even. It tries to access ‘method\_builder’ from instance self so \_\_getattribute\_\_() is called again which involves calling hasattr() again and then progressing to the next try to access ‘method\_builder’. Can you see what is next? Of course, another call to \_\_getattribute\_\_()!

So, what is the correct way to implement this by using \_\_getattribute\_\_() instead of \_\_getattr\_\_()? The answer, below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def \_\_getattribute\_\_(self, name):    try:      method = super(RESTConn, self).\_\_getattribute\_\_(name)    except AttributeError:      method = super(RESTConn, self).\_\_getattribute\_\_('method\_builder')(name)      setattr(self, name, method)      return method |