<https://docs.python.org/3/reference/datamodel.html#customizing-class-creation>

3.3.2. Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of x.name) for class instances.

object.\_\_getattr\_\_(self, name)

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 over attribute access.

object.\_\_getattribute\_\_(self, name)

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).

Note:

This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See Special method lookup.

object.\_\_setattr\_\_(self, name, value)

Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). name is the attribute name, value is the value to be assigned to it.

If \_\_setattr\_\_() wants to assign to an instance attribute, it should call the base class method with the same name, for example, object.\_\_setattr\_\_(self, name, value).

object.\_\_delattr\_\_(self, name)

Like \_\_setattr\_\_() but for attribute deletion instead of assignment. This should only be implemented if del obj.name is meaningful for the object.

object.\_\_dir\_\_(self)

Called when dir() is called on the object. A sequence must be returned. dir() converts the returned sequence to a list and sorts it.

3.3.2.1. Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called descriptor class) appears in an owner class (the descriptor must be in either the owner’s class dictionary or in the class dictionary for one of its parents). In the examples below, “the attribute” refers to the attribute whose name is the key of the property in the owner class’ \_\_dict\_\_.

object.\_\_get\_\_(self, instance, owner)

Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). owner is always the owner class, while instance is the instance that the attribute was accessed through, or None when the attribute is accessed through the owner. This method should return the (computed) attribute value or raise an AttributeError exception.

object.\_\_set\_\_(self, instance, value)

Called to set the attribute on an instance instance of the owner class to a new value, value.

object.\_\_delete\_\_(self, instance)

Called to delete the attribute on an instance instance of the owner class.

The attribute \_\_objclass\_\_ is interpreted by the inspect module as specifying the class where this object was defined (setting this appropriately can assist in runtime introspection of dynamic class attributes). For callables, it may indicate that an instance of the given type (or a subclass) is expected or required as the first positional argument (for example, CPython sets this attribute for unbound methods that are implemented in C).

3.3.2.2. Invoking Descriptors

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol: \_\_get\_\_(), \_\_set\_\_(), and \_\_delete\_\_(). If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object’s dictionary. For instance, a.x has a lookup chain starting with a.\_\_dict\_\_['x'], then type(a).\_\_dict\_\_['x'], and continuing through the base classes of type(a) excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called.

The starting point for descriptor invocation is a binding, a.x. How the arguments are assembled depends on a:

Direct CallThe simplest and least common call is when user code directly invokes a descriptor method: x.\_\_get\_\_(a).Instance BindingIf binding to an object instance, a.x is transformed into the call: type(a).\_\_dict\_\_['x'].\_\_get\_\_(a, type(a)).Class BindingIf binding to a class, A.x is transformed into the call: A.\_\_dict\_\_['x'].\_\_get\_\_(None, A).Super BindingIf a is an instance of super, then the binding super(B, obj).m() searches obj.\_\_class\_\_.\_\_mro\_\_ for the base class A immediately preceding B and then invokes the descriptor with the call: A.\_\_dict\_\_['m'].\_\_get\_\_(obj, obj.\_\_class\_\_).

For instance bindings, the precedence of descriptor invocation depends on the which descriptor methods are defined. A descriptor can define any combination of \_\_get\_\_(), \_\_set\_\_() and \_\_delete\_\_(). If it does not define \_\_get\_\_(), then accessing the attribute will return the descriptor object itself unless there is a value in the object’s instance dictionary. If the descriptor defines \_\_set\_\_() and/or \_\_delete\_\_(), it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both \_\_get\_\_() and \_\_set\_\_(), while non-data descriptors have just the \_\_get\_\_() method. Data descriptors with \_\_set\_\_() and \_\_get\_\_() defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including staticmethod() and classmethod()) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The property() function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

3.3.2.3. \_\_slots\_\_

By default, instances of classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.

The default can be overridden by defining \_\_slots\_\_ in a class definition. The \_\_slots\_\_ declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because \_\_dict\_\_ is not created for each instance.

object.\_\_slots\_\_

This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. If defined in a class, \_\_slots\_\_ reserves space for the declared variables and prevents the automatic creation of \_\_dict\_\_ and \_\_weakref\_\_ for each instance.

3.3.2.3.1. Notes on using \_\_slots\_\_

•When inheriting from a class without \_\_slots\_\_, the \_\_dict\_\_ attribute of that class will always be accessible, so a \_\_slots\_\_ definition in the subclass is meaningless.

•Without a \_\_dict\_\_ variable, instances cannot be assigned new variables not listed in the \_\_slots\_\_ definition. Attempts to assign to an unlisted variable name raises AttributeError. If dynamic assignment of new variables is desired, then add '\_\_dict\_\_' to the sequence of strings in the \_\_slots\_\_ declaration.

•Without a \_\_weakref\_\_ variable for each instance, classes defining \_\_slots\_\_ do not support weak references to its instances. If weak reference support is needed, then add '\_\_weakref\_\_' to the sequence of strings in the \_\_slots\_\_ declaration.

•\_\_slots\_\_ are implemented at the class level by creating descriptors (Implementing Descriptors) for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by \_\_slots\_\_; otherwise, the class attribute would overwrite the descriptor assignment.

•The action of a \_\_slots\_\_ declaration is limited to the class where it is defined. As a result, subclasses will have a \_\_dict\_\_ unless they also define \_\_slots\_\_ (which must only contain names of any additional slots).

•If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.

•Nonempty \_\_slots\_\_ does not work for classes derived from “variable-length” built-in types such as int, bytes and tuple.

•Any non-string iterable may be assigned to \_\_slots\_\_. Mappings may also be used; however, in the future, special meaning may be assigned to the values corresponding to each key.

•\_\_class\_\_ assignment works only if both classes have the same \_\_slots\_\_.