



UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

**Centro de
e-Learning**

UNIDAD DIDÁCTICA III

DIPLOMATURA EN PYTHON

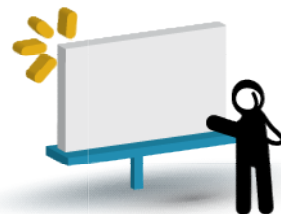
Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

www.sceu.frba.utn.edu.ar/e-learning

Módulo V – Nivel Avanzado

Unidad III – Manipulación de atributos.



Presentación:

Los atributos son nombres accesibles por los objetos de una clase que podemos definir dentro de la clase, heredarlos o crearlos dinámicamente. La forma de utilizarlos por un objeto es mediante la notación de punto, por ejemplo podríamos hacer que un objeto acceda al atributo “color” de la clase a la cual pertenece así: `objeto.color`.

En el transcurso de esta unidad veremos cómo manipular los objetos valiéndonos de diferentes técnicas y del uso de operadores.



Objetivos:

Que los participantes:

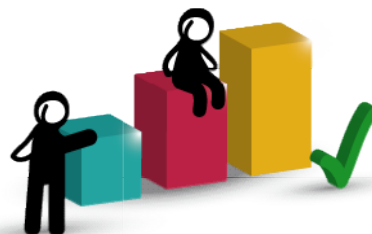
Aprendan a manipular atributos.

Analicen casos que puedan introducir problemas de loops.



Bloques temáticos:

- 1.- Introducción – Manipulación de atributos.
- 2.- Property (2.x con object y 3.x)
- 3.- Descriptores (2.x con object y 3.x)
4. `__getattr__` y `__getattribute__`, `__setattr__`, `__delattr__`



Consignas para el aprendizaje colaborativo

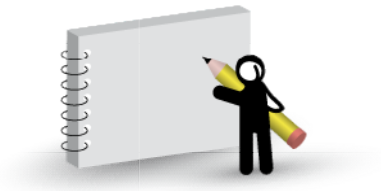
En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.



1. Introducción - Manipulación de atributos.

Por lo general los atributos son simples nombres usados para objetos, a partir de python 3.x tenemos herramientas para acceder a los atributos de forma de ejecutar un código de forma automática cada vez que el atributo es llamado. Esto podemos realizarlo de varias maneras, podemos utilizar:

- 1) `property` (solo en el nuevo estilo)
- 2) `descriptor` (solo en el nuevo estilo)
- 3) `__getattr__` y `__setattr__` (disponible en todas las versiones de python)
- 4) `__getattribute__` (solo en el nuevo estilo)

Analicemos cada uno de estas opciones.

2.- Property (2.x con object y 3.x)

Es un mecanismo que nos provee una forma de las clases del nuevo estilo para definir llamadas a métodos de forma automática para acceder o asignar atributos de instancia. Similar a los getters y setters de otros lenguajes como JAVA.

La propiedad es creada mediante la asignación del resultado de una función del built-in a un atributo de clase:

`atributo = property(fget, fset, fdel, doc)`

Ningunos de los argumentos son obligatorios, y en caso de no indicarlos se consideran que son **None**.



Veamos un primer ejemplo:

propiedades1.py

```
1 class Cliente:
2
3     def __init__(self, usuario):
4         self._usuario = usuario
5
6     def getUsuario(self):
7         print('Recupera el usuario...')
8         return self._usuario
9
10    def setUsuario(self, valor):
11        print('Modifica el usuario...')
12        self._usuario = valor
13
14    def delUsuario(self):
15        print('Remueve el usuario...')
16        del self._usuario
17    usuario = property(getUsuario, setUsuario, delUsuario, "Datos adicionales")
18
19    cliente1 = Cliente('Juan')
20    print(cliente1.usuario)
21    cliente1.usuario = 'Pedro'
22    print(cliente1.usuario)
23    del cliente1.usuario
24    print(Cliente.usuario.__doc__)
```

Al ejecutarlo obtenemos:

```
Recupera el usuario...
Juan
Modifica el usuario...
Recupera el usuario...
Pedro
Remueve el usuario...
Datos adicionales
```



Herencia de propiedades

Como los atributos, las propiedades pueden ser heredadas. Notar que al atrapar el usuario podemos actuar sobre el valor obtenido, en este caso podemos ver como en la línea 8 transformamos el nombre a mayúscula.

propiedades2.py

```
1 class MiEmpresa:
2
3     def __init__(self, usuario):
4         self._usuario = usuario
5
6     def getUsuario(self):
7         print('Recupera el usuario...')
8         return self._usuario.upper()
9
10    def setUsuario(self, valor):
11        print('Modifica el usuario...')
12        self._usuario = valor
13
14    def delUsuario(self):
15        print('Remueve el usuario...')
16        del self._usuario
17    usuario = property(getUsuario, setUsuario, delUsuario, "Datos adicionales")
18
19 class Cliente(MiEmpresa): pass
20
21 cliente1 = Cliente('Juan')
22 print(cliente1.usuario)
23 cliente1.usuario = 'Pedro'
24 print(cliente1.usuario)
25 del cliente1.usuario
26 print(Cliente.usuario.__doc__)
```

Al ejecutarlo obtenemos:

```
Recupera el usuario...
JUAN
Modifica el usuario...
Recupera el usuario...
PEDRO
```

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Remueve el usuario...
Datos adicionales

Uso de propiedades con decoradores

Podemos escribir el código que venimos trabajando de una forma un poco más intuitivo al utilizar un recurso que analizaremos un poco más adelante en este curso, el cual es el uso de decoradores. Un decorador hasta el momento para nosotros será un símbolo de arroba seguido de una palabra que antepondremos a la definición de una función. Si refactorizamos nuestro código utilizando decoradores nos queda:

propiedades2.py

```
1 class Cliente:
2
3     def __init__(self, usuario):
4         self._usuario = usuario
5
6     @property
7     def usuario(self):
8         "Datos adicionales"
9         print('Recupera el usuario...')
10        return self._usuario.upper()
11
12    @usuario.setter
13    def usuario(self, valor):
14        print('Modifica el usuario...')
15        self._usuario = valor
16
17    @usuario.deleter
18    def usuario(self):
19        print('Remueve el usuario...')
20        del self._usuario
21
22    cliente1 = Cliente('Juan')
23    print(cliente1.usuario)
24    cliente1.usuario = 'Pedro'
25    print(cliente1.usuario)
26    del cliente1.usuario
27    print(Cliente.usuario.__doc__)
```

Al ejecutarlo obtenemos:

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Recupera el usuario...

JUAN

Modifica el usuario...

Recupera el usuario...

PEDRO

Remueve el usuario...

Datos adicionales



3.- Descriptores (2.x con object y 3.x)

Los descriptores proveen una forma alternativa de interceptar atributos. En realidad una propiedad es un caso específico de descriptor. Un descriptor nos permite crear un objeto de instancia de una clase separada para proveer al atributo de las operaciones de get, set y delete.

La estructura básica de un descriptor es la siguiente:

```
class Descriptor:
    def __get__(self, instance, owner): ...
    def __set__(self, instance, value): ...
    def __delete__(self, instance): ...
```

En este caso `__get__` recibe un parámetro extra “owner”, el que indica la clase a la cual la instancia del descriptor se encuentra asociada, por lo que:

- **self**: representa la instancia de descriptor invocado por el atributo “**atributo**”
- **instance**: es en este caso la instancia de la clase (en el siguiente ejemplo “**cliente1**”)
- **owner**: Es la clase que utiliza el descriptor, (en el siguiente ejemplo “**Cliente**”)

descriptor1.py

```
1 class DescriptorUsuario:
2
3     def __get__(self, instance, owner):
4         print(self, instance, owner, sep='\n')
5
6 class Cliente:
7
8     usuario = DescriptorUsuario()
9
10 cliente1 = Cliente()
11 cliente1.usuario
12 print('-'*15)
13 Cliente.usuario
```

Al ejecutarlo obtenemos:

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

www.sceu.frba.utn.edu.ar/e-learning



```
<__main__.DescriptorUsuario object at 0x008FF6B0>
<__main__.Cliente object at 0x0095E810>
<class '__main__.Cliente'>
-----
<__main__.DescriptorUsuario object at 0x008FF6B0>
None
<class '__main__.Cliente'>
```

Nota: Si no existe una instancia de la clase que utiliza el descriptor, como en el caso en el cual la clase `DescriptorUsuario` invoca de forma directa al atributo que genera una instancia del decorador, python nos lo indica retornándonos “**None**” en su lugar.

Asignación en la instancia.

A diferencia de las propiedades, si omitimos `__set__` le permite al nombre de atributo del descriptor ser asignado o redefinido en la instancia. En el siguiente ejemplo en donde se omite `__set__` el atributo asignado a “cliente1.usuario” almacena “usuario” en la instancia del objeto “cliente1”, ocultando de este modo el descriptor almacenado en la clase `Cliente`. Esta es la forma en la cual toda asignación de un atributo de instancia trabaja en python y le permite a las clases sobrescribir de forma selectiva los niveles por defecto de clases en sus instancias.

descriptor2.py

```
1 class DescriptorUsuario:
2
3     def __get__(*args):
4         print('get')
5
6 class Cliente:
7
8     usuario = DescriptorUsuario()
9
10 cliente1 = Cliente()
11 cliente1.usuario
12 Cliente.usuario
13 cliente1.usuario = 'Juan'
14 print(cliente1.usuario)
15 print(list(cliente1.__dict__.keys()))
```

Al ejecutarlo obtenemos:

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

www.sceu.frba.utn.edu.ar/e-learning



```
get
get
Juan
['usuario']
```

Atributo de descriptor de solo lectura.

Para hacer que un atributo de descriptor sea de solo lectura, atrapamos la asignación en la clase descriptora y lanzamos una excepción para evitar la asignación del elemento.

descriptor3.py

```
1 class DescriptorUsuario:
2
3     def __get__(*args):
4         print('get')
5
6     def __set__(*args):
7         raise AttributeError('No se puede realizar un set')
8
9 class Cliente:
10
11     usuario = DescriptorUsuario()
12
13 cliente1 = Cliente()
14 cliente1.usuario
15 Cliente.usuario
16 cliente1.usuario = 'Juan'
```

Al ejecutarlo obtenemos:

```
get
get
Traceback (most recent call last):
  File "C:\Users\juanb\descriptor3.py", line 16, in <module>
    cliente1.usuario = 'Juan'
  File "C:\Users\juanb\descriptor3.py", line 7, in __set__
    raise AttributeError('No se puede realizar un set')
AttributeError: No se puede realizar un set
```



Ejemplo con descriptor fuera de la clase.

El siguiente código define un descriptor que intercepta el acceso al atributo “usuario” en sus clientes. Estos métodos usan el argumento instancia para acceder a la información en la instancia en donde el string “usuario” se encuentra almacenado.

descriptor4.py

```
1  class DescriptorUsuario:
2
3      "Documentación de descriptor de nombre"
4      def __get__(self, instance, owner):
5          print('Atrapa valor... ')
6          return instance._usuario.upper()
7
8      def __set__(self, instance, valor):
9          print('Cambia el valor ...')
10         instance._usuario = valor
11
12     def __delete__(self, instance):
13         print('Remover el atributo ...')
14         del instance._usuario
15
16 class Cliente:
17     def __init__(self, usuario):
18         self._usuario = usuario
19
20     usuario = DescriptorUsuario()
21
22 cliente1 = Cliente('Juan')
23 print(cliente1.usuario)
24 cliente1.usuario = 'Pedro'
25 print(cliente1.usuario)
26 del cliente1.usuario
27 print(cliente1.usuario)
```

Al ejecutarlo obtenemos:

```
Atrapa valor...
JUAN
Cambia el valor ...
Atrapa valor...
```

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



```
PEDRO
Remove el atributo ...
Atrapa valor...
Traceback (most recent call last):
  File "C:\Users\juanb\descriptor4.py", line 27, in <module>
    print(cliente1.usuario)
  File "C:\Users\juanb\descriptor4.py", line 6, in __get__
    return instance._usuario.upper()
AttributeError: 'Cliente' object has no attribute '_usuario'
```

Ejemplo con descriptor dentro de la clase.

Cuando no es útil poner la clase descriptor fuera de la clase cliente, es perfectamente razonable poner el descriptor dentro de la siguiente forma:

descriptor5.py

```
1  class Cliente:
2      def __init__(self, usuario):
3          self._usuario = usuario
4
5      class DescriptorUsuario:
6
7          "Documentación de descriptor de nombre"
8          def __get__(self, instance, owner):
9              print('Atrapa valor... ')
10             return instance._usuario.upper()
11
12             def __set__(self, instance, valor):
13                 print('Cambia el valor ...')
14                 instance._usuario = valor
15
16             def __delete__(self, instance):
17                 print('Remove el atributo ...')
18                 del instance._usuario
19
20     usuario = DescriptorUsuario()
21
22     cliente1 = Cliente('Juan')
23     print(cliente1.usuario)
24     cliente1.usuario = 'Pedro'
25     print(cliente1.usuario)
```



```
26 del cliente1.usuario
27 try:
28     print(cliente1.usuario)
29 except:
30     print('El usuario ya no existe')
31 print(Cliente.DescriptorUsuario.__doc__)
```

Al ejecutarlo obtenemos:

```
Atrapa valor...
JUAN
Cambia el valor ...
Atrapa valor...
PEDRO
Remove el atributo ...
Atrapa valor...
El usuario ya no existe
Documentación de descriptor de nombre
```

Dos comentarios:

1. El descriptor debe ponerse antes que la instancia del Descriptor.
2. Para acceder a la documentación del descriptor es necesario anteponer el nombre de la clase que contiene a la clase descriptora, en este caso quedaría:

Cliente.DescriptorUsuario.__doc__



Descriptor y atributos de clientes.

También es posible para un Descriptor almacenar o usar atributos asociados a la instancia de la clase cliente (atributos de instancia de la clase cliente)

descriptor6.py

```
1 class AccederInstanciaMail:
2     def __get__(self, instance, owner):
3         print('Obtener Estado de Instancia')
4         return instance._mail + '.ar'
5     def __set__(self, instance, valor):
6         print('Seteo de Estado de Instancia')
7         instance._mail = valor
8
9
10 class Cliente:
11     def __init__(self, usuario, _mail):
12         self._usuario = usuario
13         self._mail = _mail
14
15 class DescriptorUsuario:
16
17     "Documentación de descriptor de nombre"
18     def __get__(self, instance, owner):
19         print('Atrapa valor... ')
20         return instance._usuario.upper()
21
22     def __set__(self, instance, valor):
23         print('Cambia el valor ...')
24         instance._usuario = valor
25
26     def __delete__(self, instance):
27         print('Remover el atributo ...')
28         del instance._usuario
29
30     usuario = DescriptorUsuario()
31     mail = AccederInstanciaMail()
32
33
34 cliente1 = Cliente('Juan', 'juan@gmail.com')
35 print(cliente1.usuario, cliente1._mail, cliente1.mail)
36 cliente1.usuario = 'Ana'
```



```
37 print(cliente1.usuario)
38 cliente1.mail = 'ana@gmail.com'
39 print(cliente1.usuario, cliente1._mail, cliente1.mail)
```

Al ejecutarlo obtenemos:

```
Atrapa valor...
Obtener Estado de Instancia
JUAN juan@gmail.com juan@gmail.com.ar
Cambia el valor ...
Atrapa valor...
ANA
Seteo de Estado de Instancia
Atrapa valor...
Obtener Estado de Instancia
ANA ana@gmail.com ana@gmail.com.ar
```

En este caso “mail” es asignada a un descriptor. El nuevo descriptor no tiene información por sí mismo, pero utiliza un atributo que asume que existe en la clase cliente llamado “_mail” para evitar colisión con el nombre del descriptor en sí. Si solicitamos la siguiente impresión:

```
print(cliente1.__dict__)
```

Podemos ver los atributos de instancia y sus valores.

```
{'_usuario': 'Ana', '_mail': 'ana@gmail.com'}
```



Almacenar datos de instancia y de descriptor.

Notar que en este caso tanto el atributo de instancia de la clase cliente como del Descriptor se llaman igual "mail", y se almacenan los dos valores. Pero al recuperar los datos de la instancia de la clase cliente con `__dict__` como es de esperar solo trae el valor almacenado en la instancia.

descriptor7.py

```
1 class AccederInstanciaMail:
2
3     def __init__(self, mail):
4         self.mail = mail
5     def __get__(self, instance, owner):
6         return '%s, %s' % (self.mail, instance.mail)
7     def __set__(self, instance, valor):
8         instance.mail = valor
9
10 class Cliente:
11
12     def __init__(self, mail):
13         self.mail = mail
14         administradormail = AccederInstanciaMail('ana@gmail.com')
15
16 cliente1 = Cliente('juan@gmail.com')
17 print(cliente1.administradormail)
18 cliente1.administradormail = 'juanbarreto@gmail.com'
19 print(cliente1.administradormail)
20 print('-'*10)
21 print(cliente1.__dict__)
```

Al ejecutarlo obtenemos:

```
ana@gmail.com, juan@gmail.com
ana@gmail.com, juanbarreto@gmail.com
-----
{'mail': 'juanbarreto@gmail.com'}
```



Para versiones mayores o igual a la 3.6

Existe un nuevo método:

`__set_name__(self, owner, name)`

Con este nuevo método, cada vez que se crea una instancia de un descriptor, se llama a este método y el parámetro de nombre se establece automáticamente. Esto hace posible crear su descriptor sin especificar el nombre del atributo interno que se necesita usar para almacenar el valor.

`descriptor36_o_mayor.py`

```
class Descriptor():
    def __set_name__(self, owner, nombre):
        self.nombre = nombre

    def __get__(self, instancia, type=None) -> object:
        return instancia.__dict__.get(self.nombre) or 0

    def __set__(self, instancia, valor) -> None:
        instancia.__dict__[self.nombre] = valor

class Clase():
    atributo_descriptor = Descriptor()

objeto1 = Clase()
objeto2 = Clase()
objeto1.atributo_descriptor = 3
print(objeto1.atributo_descriptor)
print(objeto2.atributo_descriptor)
```



4. `__getattr__` y `__getattribute__`, `__setattr__`, `__delattr__`

Estos métodos también se pueden utilizar para interceptar atributos.

`__getattr__` : Se ejecuta para atributos indefinidos ya que es utilizado para atributos no almacenados en un atributo de instancia o heredado de otra clase. Su uso es muy sencillo. **(No requiere utilizar el nuevo estilo)**(Solo se utiliza con atributos indefinidos.)

`__getattribute__` : Es ejecutado para cada atributo. Hay que tener cuidado cuando utilizamos este método para no sufrir un problema de loops de recursividad mediante acceso de atributos a una superclase. **(Requiere utilizar el nuevo estilo)**(Se puede utilizar con todos los atributos) (Hay que tener cuidado con la generación de loops)

Ejemplo

Estos métodos permiten atrapar, mostrar y setear atributos, vamos a crear una clase que en el caso de que creamos un atributo “color” lo capture con `__getattr__` y retorne un color aleatorio rgb para la web. En el caso de que el atributo no sea “color” solamente guardamos el valor en el diccionario de la instancia:

descriptor7.py

```
1 from random import randint
2
3 class Auto:
4
5     def __getattr__(self, dato):
6         if dato == 'color':
7             rojo = randint(0, 255)
8             verde = randint(0, 255)
9             azul = randint(0, 255)
10            return 'rgb('+str(rojo)+' '+str(verde)+' '+str(azul)+')'
11        else:
12            raise AttributeError(dato)
13
14    def __setattr__(self, dato, valor):
15        print('set: %s %s' % (dato, valor))
16        if dato == 'color':
17            self.__dict__['_color'] = valor
18
```



```
19     else:
20         self.__dict__[dato] = valor
21
22     auto1 = Auto()
23     print(auto1.color)
24     auto1.color = '(f,0,0)'
25     print('-----')
26     print(auto1._color)
27     print('-----')
28     print(auto1.color)
29     print(auto1.color)
30     print(auto1.color)
31     print('-----')
32     print(auto1._color)
33     print('-----')
34
35     auto1.altura = '1.6 metros'
36     print(auto1.altura)
37     print('-----')
38     print(auto1.__dict__)
```

Al ejecutarlo obtenemos:

```
rgb(33,188,80)
set: color (f,0,0)
-----
(f,0,0)
-----
rgb(221,31,104)
rgb(32,38,252)
rgb(29,5,101)
-----
(f,0,0)
-----
set: altura 1.6 metros
1.6 metros
-----
{'_color': '(f,0,0)', 'altura': '1.6 metros'}1.6 metros
-----
{'_color': '(f,0,0)', 'altura': '1.6 metros'}
```




Problemas con loops en `__getattribute__`

Al utilizar `__getattribute__` o `__setattr__` podemos crear un loop indefinido que puede generar la saturación de la memoria. Veamos cómo solucionarlo.

CASO DE `__getattribute__`

En el caso de `__getattribute__` el siguiente código crearía un loop ya que cuando se llame a `"self.color"` se ejecutaría nuevamente el método.

`getattr_setattr_loop.py`

```
1 class Auto:
2     def __init__(self):
3         self.color = 'Rojo'
4
5     def __getattribute__(self, color):
6         return self.color
7
8 auto1 = Auto()
9 print(auto1.color)
```

Al ejecutarlo obtenemos:

```
Traceback (most recent call last):
  File "C:\Users\juanb\getattr-setattr-loop.py", line 9, in <module>
    print(auto1.color)    #return object.__getattribute__(self, color)
  File "C:\Users\juanb\getattr-setattr-loop.py", line 6, in __getattribute__
    return self.color
  File "C:\Users\juanb\getattr-setattr-loop.py", line 6, in __getattribute__
    return self.color
  File "C:\Users\juanb\getattr-setattr-loop.py", line 6, in __getattribute__
    return self.color
[Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

La forma de evitarlo es pasar el objeto atrapado a una clase superior, como puede ser `object`, para evitar de esta forma la recursividad



getattr_setattr_loop.py

```
1 class Auto:
2     def __init__(self):
3         self.color = 'Rojo'
4
5     def __getattr__(self, color):
6         return object.__getattr__(self, color)
7
8 auto1 = Auto()
9 print(auto1.color)
```

Al ejecutarlo obtenemos:

Rojo

Problemas con loops en __setattr__

Al usar setattr podemos tener un problema de loops si en la línea siguiente cambiamos:

self.__dict__[atributo] = valor + 202000

por:

self.atributo = valor + 202000

getattr_setattr_loop2.py

```
1 class Auto:
2     def __setattr__(self, atributo, valor):
3         if atributo == 'precio':
4             self.__dict__[atributo] = valor + 202000
5             #self.atributo = valor + 202000
6
7         else:
8             raise Auto(atributo + ' No permitido')
9 auto1 = Auto()
10 auto1.precio = 1000000
11 print(auto1.precio)
```

Al ejecutarlo obtenemos:

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

www.sceu.frba.utn.edu.ar/e-learning



1202000

Uso correcto de __getattr__, __setattr__ y __delattr__

getattr_setattr_delattr.py

```
1 class Persona:
2
3     def __init__(self, nombre):
4         self._nombre = nombre
5
6     def __getattr__(self, atributo):
7         print('get: ' + atributo)
8         if atributo == 'nombre':
9             return self._nombre
10        else:
11            raise AttributeError(atributo)
12
13    def __setattr__(self, atributo, valor):
14        print('set: ' + atributo)
15        if atributo == 'nombre':
16            atributo = '_nombre'
17        self.__dict__[atributo] = valor
18
19    def __delattr__(self, atributo):
20        print('del: ' + atributo)
21        if atributo == 'nombre':
22            atributo = '_nombre'
23        del self.__dict__[atributo]
24
25    persona1 = Persona('Juan')
26    print(persona1.nombre)
27    persona1.nombre = 'Pedro'
28    del persona1.nombre
```

Al ejecutarlo obtenemos:

```
set: _nombre
get: nombre
Juan
set: nombre
del: nombre
```

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Bibliografía utilizada y sugerida

Libros

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

Manual online

<https://docs.python.org/3.7/tutorial/>

<https://docs.python.org/3.7/library/index.html>

<https://docs.python.org/3/reference/datamodel.html>



Lo que vimos

En esta unidad hemos trabajado sobre la manipulación de atributos.



Lo que viene:

En la siguiente unidad comenzaremos a trabajar con decoradores.