Lenguaje de Programación Python

Soporte Orientado a Objetos: Porpiedades

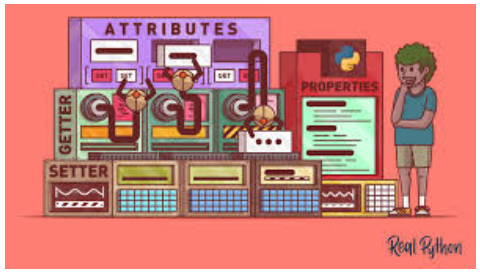
Dr. Mario Marcelo Berón
Argentina Programa
Universidad Nacional de San Luis

Índice

Propiedades

Con property() se puede crear atributos administrados en las clases.

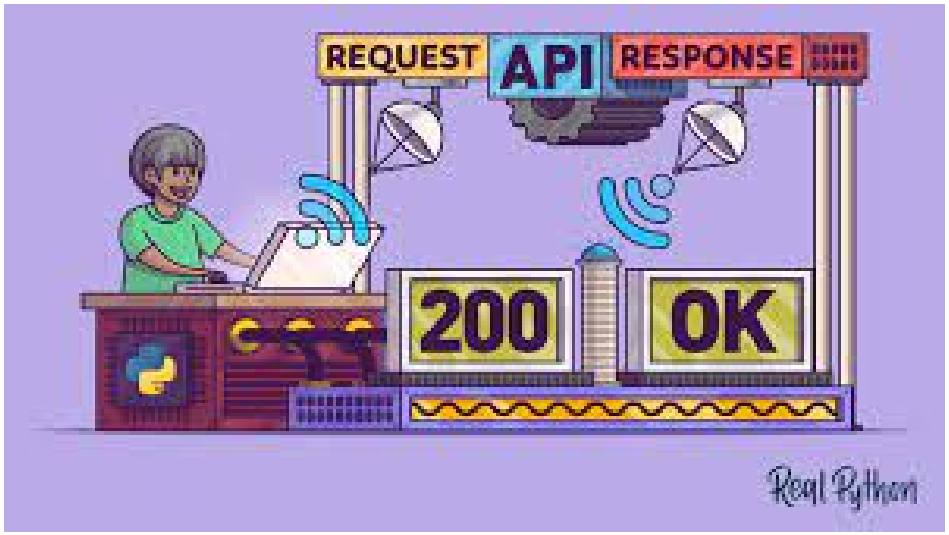
Se pueden utilizar cuando se necesita modificar la implementación interna sin cambiar la API pública de la clase.



Podría decirse que las propiedades es la forma más popular de crear atributos administrados rápidamente y al más puro estilo pitónico.

Administración de Atributos en las Clases

Cuando se define una clase en un lenguaje de programación orientado a objetos, probablemente se necesitarán atributos de clase e instancia.



Los atributos mantienen el estado interno de un objeto determinado, al que a menudo necesitará acceder y mutar.

Administración de Atributos en las Clases

Por lo general, se tienen dos formas de administrar un atributo

Acceder al atributo directamente.

Usar métodos.

Importante

Si se exponen los atributos al usuario, se vuelven parte de la API pública de la clase. El usuario accederá y los mutará directamente en su código. El problema surge cuando se necesita cambiar la implementación interna de un atributo. **Algunos programas que usan la clase dejarán de funcionar!**.

Administración de Atributos en las Clases

Comentarios

Lenguajes de programación como Java y C++ alientan a no exponer los atributos para evitar este tipo de problema. En su lugar, se deben proporcionar métodos getter y setter, también conocidos como observadores y modificadores, respectivamente. Estos métodos ofrecen una forma de cambiar la implementación interna de sus atributos sin cambiar su API pública.

Administración de Atributos en las Clases-getters y setters

```
class Punto:
 def __init__(self, x, y):
  self._x = x
  self._y = y

 def get_x(self):
  return self._x

 def set_x(self, valor):
  self._x = valor

 def get_y(self):
  return self._y

 def set_y(self, valor):
  self._y = valor
```

En este ejemplo, se crea Punto con dos atributos no públicos ._x y ._y los cuales contienen las coordenadas cartesianas del punto en cuestión.

Administración de Atributos en las Clases-getters y setters

```
class Punto:
 def __init__(self, x, y):
  self._x = x
  self._y = y

 def get_x(self):
  return self._x

 def set_x(self, valor):
  self._x = valor

 def get_y(self):
  return self._y

 def set_y(self, valor):
  self._y = valor
```

Para acceder y mutar el valor de ._x o ._y, puede usar los métodos getter y setter correspondientes.

Administración de Atributos en las Clases-getters y setters

```
>>> from punto import Punto

>>> punto = Punto(12, 5)
>>> punto.get_x()
12
>>> punto.get_y()
5
>>> punto.set_x(42)
>>> punto.get_x()
42
>>> # Los atributos no
#públicos son accesibles
>>> punto._x
42
>>> punto._y
5
```

Con .get_x() y .get_y(), puede acceder a los valores actuales de ._x y ._y. Se puede usar el método setter para almacenar un nuevo valor en el atributo administrado correspondiente. Se puede confirmar que Python no restringe el acceso a atributos no públicos.

Administración de Atributos en las Clases-getters y setters

```
>>> class Punto:
...
def __init__(self, x, y):
...             self.x = x
...             self.y = y
...

>>> punto = Punto(12, 5)
>>> punto.x
12
>>> punto.y
5

>>> punto.x = 42
>>> punto.x
42
```

Exponer atributos al usuario final es normal y común en Python. No se necesita implementar los métodos getter y setter todo el tiempo. Sin embargo, *¿cómo se pueden manejar los cambios en los requisitos que implican cambios en la API?*

Administración de Atributos en las Clases-getters y setters

Comentario

A diferencia de Java y C++, Python proporciona herramientas útiles que permiten cambiar la implementación subyacente de los atributos sin modificar su API pública. El enfoque más popular es convertir sus atributos en propiedades. Las propiedades representan una funcionalidad intermedia entre un atributo (o campo) y un método. En otras palabras, permiten crear métodos que se comportan como atributos.

Administración de Atributos en las Clases-Propiedades

Propiedades

property (fget=None, fset=None, fdel=None, doc=None)

- fget: función que retorna el valor de un atributo.
- fset: función que permite cambiar el valor de un atributo.
- fdel: función que define como un atributo maneja la supresión.
- doc: un docstring del atributo.

Administración de Atributos en las Clases-Propiedades

Propiedades

property (fget=None, fset=None, fdel=None, doc=None)

Comentarios

El valor de retorno de property() es el propio atributo gestionado. Si accede al atributo, como en obj.attr, Python llama automáticamente a fget(). Si asigna un nuevo valor al atributo, como en obj.attr = valor, Python llama a fset() usando el valor de entrada como argumento. Finalmente, si ejecuta una instrucción del obj.attr, Python llama automáticamente a fdel().

Administración de Atributos en las Clases-Propiedades

Propiedades

property (fget=None, fset=None, fdel=None, doc=None)

comentarios

Se puede usar property() ya sea como una función o un decorador para implementar propiedades.

Administración de Atributos en las Clases-Propiedades

```
class Círculo:
 def __init__(self, radio):
  self._radio = radio

 def _get_radio(self):
  print("Get radio")
  return self._radio

 def _set_radio(self, valor):
  print("Set radio")
  self._radio = valor

 def _del_radio(self):
  print("Delete radio")
  del self._radio
```


```
radio = property(
fget=_get_radio,
fset=_set_radio,
fdel=_del_radio,
doc="La propiedad radio."
)
```

En este fragmento de código, se crea la clase Círculo. El inicializador de clase .__init__(), toma el radio como argumento y lo almacena en un atributo no público llamado ._radio.

Argentina programa 4.0

Administración de Atributos en las Clases-Propiedades

```
class Círculo:
 def __init__(self, radio):
  self._radio = radio

 def _get_radio(self):
  print("Get radio")
  return self._radio

 def _set_radio(self, valor):
  print("Set radio")
  self._radio = valor

 def _del_radio(self):
  print("Delete radio")
  del self._radio
```

Luego define tres métodos no públicos:

1. ._get_radio() devuelve el valor actual de *._radio*.
2. ._set_radio() toma valor como argumento y lo asigna a *._radio*.
3. ._del_radio() elimina el atributo de instancia *._radio*.

Administración de Atributos en las Clases-getters y setters

Uso de Círculo con Propiedades

```
>>> from círculo import Círculo
>>> círculo = Círculo(42.0)
>>> círculo.radio
Get radio
42.0
>>> círculo.radio = 100.0
Set radio
```

Administración de Atributos en las Clases-getters y setters

Uso de Círculo con Propiedades

```
>>> from círculo import Círculo
.....
>>> círculo.radio
Get radio
100.0
>>> del círculo.radio
Delete radio
```

Administración de Atributos en las Clases-getters y setters

Comentario

La propiedad *.radio* oculta la variable de instancia privada *._radio*, que en este ejemplo es un atributo administrado. Se puede acceder y asignar *.radio* directamente. Internamente, Python llama automáticamente a *._get_radio()* y *._set_radio()* cuando sea necesario. Cuando se ejecuta *del circulo.radio*, Python llama a *._del_radio()*, que elimina el *._radio* subyacente.

Administración de Atributos en las Clases-property() como Decorador

Decorador

Los *decoradores* están en todas partes en Python. Son funciones que toman otra función como argumento y devuelven una nueva función con funcionalidad añadida. Con un decorador, se puede adjuntar operaciones de procesamiento previo y posterior a una función existente.

Administración de Atributos en las Clases-property() como Decorador

Decorador - Sintaxis

```
@decorador
 def func(a):
  return a
```

Administración de Atributos en las Clases-property() como Decorador

Decorador - Sintaxis

```
def func(a):
  return a

func = decorador(func)
```

La última línea de código hace *func* contenga el resultado de llamar a *decorador(func)*.

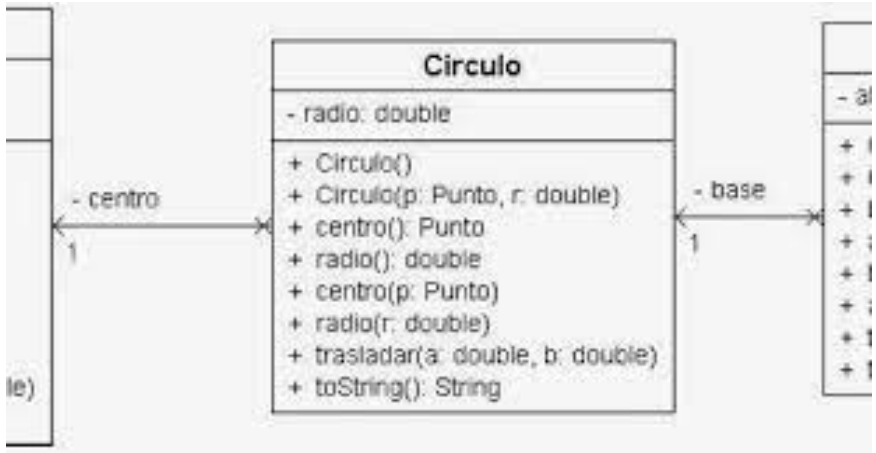
Administración de Atributos en las Clases-Propiedades con Decoradores

```
class Círculo:

    def __init__(self, radio):
        self._radio = radio

    @property
    def radio(self):
        """La Propiedad Radio."""
        print("Get radio")
        return self._radio

    @radio.setter
    def radius(self, valor):
        print("Set radio")
        self._radio = valor
```




```
    @radio.deleter
    def radio(self):
        print("Delete radio")
        del self._radio
```

Administración de Atributos en las Clases-Propiedades con Decoradores

Funcionamiento con Propiedades

```
>>> from círculo import Círculo
>>> círculo = Círculo(42.0)
>>> círculo.radio
Get radio
42.0
>>> círculo.radio = 100.0
Set radio
```

Administración de Atributos en las Clases-Propiedades con Decoradores

Funcionamiento con Propiedades

```
>>> círculo.radio
Get radio
100.0
>>> del círculo.radio
Delete radio
```

Administración de Atributos en las Clases-Propiedades con Decoradores

Funcionamiento con Propiedades

```
>>> círculo.radio
Get radio
Traceback (most recent call last):
...
AttributeError: 'Círculo' object has
 no attribute '_radio'
```

Administración de Atributos en las Clases-Propiedades con Decoradores

Funcionamiento con Propiedades

```
>>> help(círculo)
Help on Círculo in module __main__
object:

class Círculo(builtins.object)
...
|   radio
|       The radio property.
```

Administración de Atributos en las Clases-Propiedades con Decoradores

Funcionamiento con Propiedades

No se necesita usar un par de paréntesis para llamar a *.radio()* como método. En su lugar, se puede acceder a *.radio* como accedería a un atributo normal, que es el uso principal de las propiedades. Las propiedades permiten tratar los métodos como atributos y se encargan de llamar automáticamente al conjunto subyacente de métodos.

Administración de Atributos en las Clases-Atributo de Solo Lectura

```
class Punto:
 def __init__(self, x, y):
  self._x = x
  self._y = y

 @property
 def x(self):
  return self._x

 @property
 def y(self):
  return self._y
```

```
>>> punto = Punto(12, 5)
>>> # Leer cordenadas
>>> punto.x
12
>>> punto.y
5
>>> # Esc. coordenadas
>>> punto.x = 42
Traceback ...:
...
AttributeError:.....
```

Administración de Atributos en las Clases-Atributo de Solo Lectura

```
class ErrorDeEscDeCoord(Exception):
 pass

class Punto:
 def __init__(self, x, y):
  self._x = x
  self._y = y

 @property
 def x(self):
  return self._x
```

Administración de Atributos en las Clases-Atributo de Solo Lectura

```
class Punto
 @x.setter
 def x(self, value):
  raise ErrorDeEscDeCoord("x es de
    solo lectura")
@property
def y(self):
return self._y

@y.setter
def y(self, valor):
raise ErrorDeEscDeCoord("y es de solo lectura")
```

Administración de Atributos en las Clases-Atributo de Lectura-Escritura

```
import math

class Círculo:
 def __init__(self, radio):
  self.radio = radio

 @property
 def radio(self):
  return self._radio
```

Administración de Atributos en las Clases-Atributo de Lectura-Escritura

```
@radio.setter
def radio(self, valor):
 self._radio = float(valor

@property
def diámetro(self):
 return self.radio * 2

@diámetro.setter
def diámetro(self, valor):
 self.radio = valor / 2
```

Administración de Atributos en las Clases-Atributo de Solo Lectura

```
>>> from círculo import Cír

>>> círculo = Círculo(42)
>>> círculo.radio
42.0

>>> círculo.diámetro
84.0
```

Administración de Atributos en las Clases-Atributo de Lectura-Escritura

```
>>> círculo.diámetro = 100
>>> círculo.diámetro
100.0

>>> círculo.radio
50.0
```

Administración de Atributos en las Clases-Atributo de Solo Escritura

```
import hashlib
import os

class Usuario:
def __init__(self, nombre, clave):
self.nombre = nombre
self.clave  = clave
self._hashed_clave=""
```

Administración de Atributos en las Clases-Atributo de Solo Escritura

```
@property
def clave(self):
raise AttributeError("clave es de solo escritura")

@clave.setter
def clave(self, texto):
salt = os.urandom(32)
self._hashed_password = hashlib.pbkdf2_hmac(
"sha256", texto.encode("utf-8"), salt, 100_000)
```

Administración de Atributos en las Clases-Atributo de Solo Escritura

```
>>> from usuarios import Usuario

>>> juan = User("Juan", "secreto")
```

```
>>> juan._hashed_clave
b'b\xc7^ai\x9f3\xd2g␣...␣\x89^-\x92\xbe\xe6'
```

Administración de Atributos en las Clases-Atributo de Solo Escritura

```
>>> john.clave
Traceback (most recent call last):
...
AttributeError:
 Clave es de solo lectura
```

```
>>> john.calve = "super-secreto"
>>> john._hashed_clave
b'\xe9l$\x9f\xaf\x9d␣...␣b\xe8\xc8\xfcaU\r_'
```