properties

February 16, 2024

[25]:
```
# property() or @property allows us to add managed
# attributes to our class, so called properties
```

[26]:
```
# Properties allow you to define methods that behave
# like attributes.
```

[27]:
```
# Let's first use some other techniques:
```

[28]:
```
# getters and setters:

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def get_x(self):
        return self._x
    def set_x(self, x):
        self._x = x
    def get_y(self):
        return self._y
    def set_y(self, y):
        self._y = y

point = Point(3, 5)

print(point.get_x())
print(point.get_y())

point.set_x(1)
point.set_y(8)

print(point.get_x())
print(point.get_y())
```

```
3
5
1
8
```

[29]:
```
# It is common in practice in python though to just
# expose attributes publicly removing the need for
# getters and setters

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

point = Point(3,5)

print(point.x)
print(point.y)

point.x = 1
point.y = 8

print(point.x)
print(point.y)
```

```
3
5
1
8
```

[30]:
```
# If you want getters and setters, for example to attach
# more logic to it, but still want your attributes to be
# easy to reach is where properties shine.

# They essentially expose your getters and setters in a
# neat way. The best way to illustrate this is by using
# the property function itself:

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    # Non-public getters, setters and deleters:
    def _get_x(self):
        return self._x

    def _set_x(self, x):
        self._x = x

    def _del_x(self):
        del self._x
```

```
    def _get_y(self):
        return self._y

    def _set_y(self, y):
        self._y = y

    def _del_y(self):
        del self._y

    # define two properties:
    x = property(
        fget=_get_x,
        fset=_set_x,
        fdel=_del_x
    )
    y = property(
        fget=_get_y,
        fset=_set_y,
        fdel=_del_y
    )

# Usage:
point = Point(3, 5)
print(point.x)
print(point.y)
point.x = 1
point.y = 8
print(point.x)
print(point.y)

del point.x
# print(point.x) # _x is no more
# let's get it back
point.x = 1
print(point.x)
print(dir(Point.x))
```

```
3
5
1
8
1
['__class__', '__delattr__', '__delete__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__getstate__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__isabstractmethod__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__set__', '__set_name__', '__setattr__', '__sizeof__', '__str__',
```

```
'__subclasshook__', 'deleter', 'fdel', 'fget', 'fset', 'getter', 'setter']
```

[31]:
```python
# Properties are class methods that manage instance
# attributes.
```

[32]:
```python
# A more concise use involves the @property decorator:

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

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

    @x.setter
    def x(self, x):
        self._x = x

    @x.deleter
    def x(self):
        del self._x

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

    @y.setter
    def y(self, x):
        self._y = x

    @y.deleter
    def y(self):
        del self._y

# Usage:
point = Point(3, 5)
print(point.x)
print(point.y)
point.x = 1
point.y = 8
print(point.x)
print(point.y)
```

```
3
5
1
```

```
8
```

[33]:
```python
# That's nice, but what is really cool is that you are
# now able to attach additional logic to your properties,
# like access control or input validation.

# Also you can define properties that are computed at
# runtime, meaning they are computed when requested.

# Example:

class Square:
    def __init__(self, length):
        self.length = length

    @property
    def area(self):
        return self.length * self.length

square = Square(4)
print(square.area)

print("That's cool, huh?")
```

```
16
That's cool, huh?
```

[34]:
```python
# Another example incorporating input validation:
class Person:
    def __init__(self, name: str, id: int) -> None:
        self.name = name
        self._id = id

    @property
    def id(self):
        return self._id

    @id.setter
    def id(self, id):
        try:
            self._id = int(id)
        except ValueError:
            # if you want just a message:
            print("id must be an integer")
            # if you want it to break execution:
            # raise ValueError("id must be an integer")
            # hide some details that are not needed for the end-user
```

```
            #raise ValueError("id must be an integer") from None


person = Person("Simon", 1)
person.id = "my_id"
print(dir(person))
```

```
id must be an integer
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_id', 'id', 'name']
```

[35]:
```
class MyClass:
    def __init__(self, x):
        self._x = x

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

example = MyClass(42)
print(example.x)
# example.x = 8
example._x = 8
print(example.x)
```

```
42
8
```

[ ]:
```

```

[36]:
```
# Write only property:

class WriteOnly:
    def __init__(self, x):
        self._x = x
    @property
    def x(self):
        raise AttributeError("Not allowed!")
    @x.setter
    def x(self, x):
        self._x = x

write_only = WriteOnly(8)
write_only.x = 40
```

```
print(write_only.x)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[36], line 15
     13 write_only = WriteOnly(8)
     14 write_only.x = 40
---> 15 print(write_only.x)

Cell In[36], line 8, in WriteOnly.x(self)
      6 @property
      7 def x(self):
----> 8     raise AttributeError("Not allowed!")

AttributeError: Not allowed!
```

[ ]: