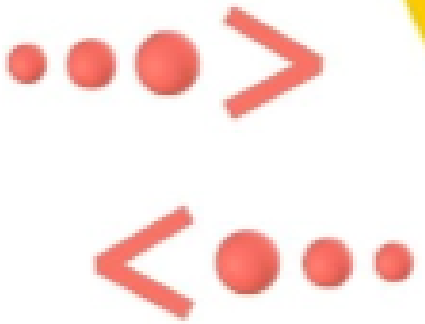
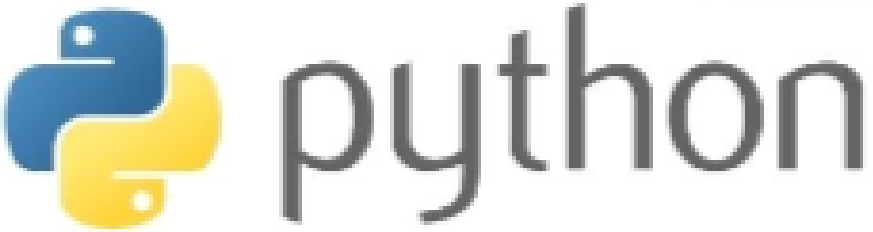
Python


@property

Getter and Setter in Python





Python Property Decorator – @property

A decorator feature in Python wraps in a function, appends several functionalities to existing code and then returns it. Methods and functions are known to be callable as they can be called. Therefore, a decorator is also a callable that returns callable. This is also known as **metaprogramming** as at compile time a section of program alters another section of the program. **Note:** For more information, refer to Decorators in Python

Python @property decorator

@property decorator is a built-in decorator in Python which is helpful in defining the properties effortlessly without manually calling the inbuilt function property(). Which is used to return the property attributes of a class from the stated getter, setter and deleter as parameters. **Now, lets see some examples to illustrate the use of @property decorator in Python: Example 1:**

```python
# Python program to illustrate the use of
# @property decorator

# Defining class
class Portal:

    # Defining __init__ method
    def __init__(self):
        self.__name =''

    # Using @property decorator
    @property

    # Getter method
    def name(self):
        return self.__name

    # Setter method
    @name.setter
    def name(self, val):
        self.__name = val

    # Deleter method
    @name.deleter
    def name(self):
       del self.__name

# Creating object
p = Portal();

# Setting name
p.name = 'GeeksforGeeks'

# Prints name
print (p.name)

# Deletes name
del p.name

# As name is deleted above this
# will throw an error
print (p.name)
```

Output:

```
GeeksforGeeks


## An error is thrown
Traceback (most recent call last):
  File "main.py", line 42, in
    print (p.name)
  File "main.py", line 16, in name
    return self.__name
AttributeError: 'Portal' object has no attribute '_Portal__name'
```

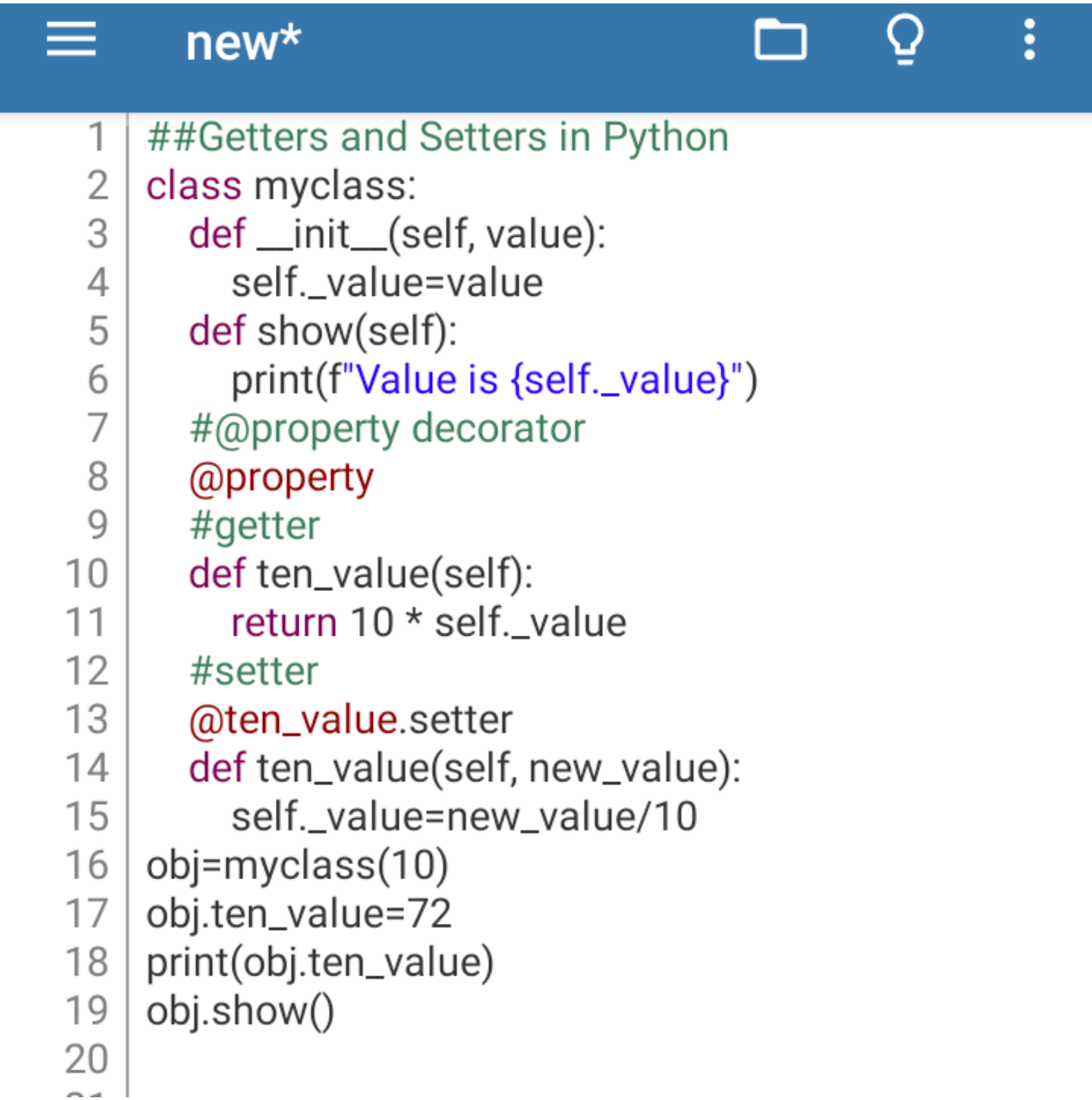
Here, the @property decorator is used to define the property name in the class Portal, that has three methods(getter, setter, and deleter) with similar names i.e, name(), but they have different number of parameters. Where, the method **name(self)** labeled with @property is a getter method, **name(self, val)** is a setter method as it is used to set the value of the attribute **__name** and so its labeled with @name.setter. Lastly, the method labeled with *@name.deleter* is a deleter method which can delete the assigned value by the setter method. However, deleter is invoked with the help of a keyword del. **Example 2:**

```python
# Python program to illustrate the use of
# @property decorator

# Creating class
class Celsius:

    # Defining init method with its parameter
    def __init__(self, temp = 0):
        self._temperature = temp

    # @property decorator
    @property

    # Getter method
    def temp(self):

        # Prints the assigned temperature value
        print("The value of the temperature is: ")
        return self._temperature

    # Setter method
    @temp.setter
    def temp(self, val):

        # If temperature is less than -273 than a value
        # error is thrown
        if val < -273:
            raise ValueError("It is a value error.")

        # Prints this if the value of the temperature is set
        print("The value of the temperature is set.")
        self._temperature = val

# Creating object for the stated class
cel = Celsius();

# Setting the temperature value
cel.temp = -270

# Prints the temperature that is set
print(cel.temp)

# Setting the temperature value to -300
# which is not possible so, an error is
# thrown
cel.temp = -300
print(cel.temp)
```

Output:

```
The value of the temperature is set.

The value of the temperature is:

-270


# An error is thrown

Traceback (most recent call last):

  File "main.py", line 47, in

    cel.temp = -300

  File "main.py", line 28, in temp

    raise ValueError("It is a value error.")

ValueError: It is a value error.
```

Here, a value error is thrown as the value of the temperature assigned must be above -273. But here it is -300. Hence, a value error is thrown.


new*
##Getters and Setters in Python
class myclass:
def __init__(self, value):
self._value=value
def show(self):
print(f"Value is {self._value}")
#@property decorator
@property
#getter
def ten_value(self):
return 10 * self._value
#setter
@ten_value.setter
def ten_value(self, new_value):
self._value=new_value/10
obj=myclass(10)
obj.ten_value=72
print(obj.ten_value)
obj.show()


TAB
72.0
Value is 7.2
[Program finished]