@property Decorator

@property decorator allows us to define properties easily without calling the `property()` function manually. Before learning about the @property decorator, let's understand what is a decorator.

What is a decorator?

In Python, the function is a first-order object. It means that it can be passed as an argument to another function. It is also possible to define a function inside another function. Such a function is called a nested function. Moreover, a function can return another function.

A decorator is a function that receives another function as argument. The behaviour of the argument function is extended by the decorator without actually modifying it.

The typical decorator function will look like below.

Decorator Function
```
def mydecoratorfunction(some_function): # function to be decorated passed as
argument
    def wrapper_function(): # wrap the some_function and extends its
behaviour
        # write code to extend the behaviour of some_function()
        some_function() # call some_function
        return wrapper_function # return wrapper function
```

In the above example, the `some_function` is a function whose behaviour we want to extend. So, we will have to write a custom function like `mydecoratorfunction()`, which takes the `some_function` as an argument. The `wrapper_function()` is an inner function where we can write additional code to extend the behaviour of the `some_function`, before or after calling it. And finally, the `wrapper_funtion()` should be returned. In this way, Python includes decorator functions. Also, we can define our own decorator function to extend the behaviour of a function without modifying it.

Now, let's take a simple example to demonstrate how to define a custom decorator function and its' usage. Consider the following simple function.

Example:
```
def display(str):
    print(str)
```

Now, let's define a decorator function which modifies the output of the above `display()` function by prepending 'Output:' to the result of the `display()` function.

Example: Decorator Function
```
def displaydecorator(fn):
    def display_wrapper(str):
        print('Output:', end=" ")
        fn(str)
    return display_wrapper
```

You can now decorate this function to extend its behaviour by passing it to the decorator.

```
>>> out=displaydecorator(display)
>>> out('Hello World')
Output: Hello World
```

As you can see, 'Output' is prepended with the result of the `display()` function. The `displaydecorator()` function is used to modify the behaviour of the `display()` function without modifying it.

Python includes the @[decorator_function_name] syntax to specify a decorator function. We can specify @displaydecorator to the `display()` function to denote that the `display()` function is decorated with the `displaydecorator()` function, as shown below.

Example: @decorator

```
@displaydecorator
def display(str):
    print(str)
```

After applying the decorator `@displaydecorator` in the above example, we can directly call the `display()` function to get the extended behaviour, as shown below.

```
>>> display('Hello World')
Output: Hello World
```

@property Decorator

@property decorator is a built-in decorator in Python for the property() function.

The following code uses the built-in @property decorator to define the `name` property in the `person` class.

Example: @property decorator

```
class person:
    def __init__(self):
        self.__name=''
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, value):
        self.__name=value
```

The above `person` class includes two methods with the same name `name()`, but with a different number of parameters. This is called method overloading. The `name(self)` function is marked with the @property decorator which indicates that the `name(self)` method is a getter method and the name of the property is the method name only, in this case `name`. Now, `name(self, value)` is assigning a value to the private attribute `__name`. So, to mark this method as a setter method for the `name` property, the `@name.setter` decorator is applied. This is how we can define a property and its getter and setter methods.

Now, we can access the `name` property without calling the getter or setter method manually, as shown below.

```
>>> p=person()
>>> p.name='Steve'
>>> p.name
Steve
```

Use @[property-name].setter to call a setter method and @[property-name].deleter to call deleter method.

Example: @property decorator

```
class person:
    def __init__(self):
        self.__name=''
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, value):
        self.__name=value
    @name.deleter
    def name(self, value):
        print('Deleting..')
        del self.__name
```

The deleter would be invoked when you delete the property using keyword `del`, as shown below.

```
>>> p=person()
>>> p.name='Steve'
>>> p.name
Steve
>>> del p.name
Deleting..
>>> p.name
Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>
p.name
File "C:\Python37\test.py", line 6, in name
```

```
return self.__name
AttributeError: 'person' object has no attribute '_person__name'
```