decorator_setter_deleter

December 3, 2022

In this section, we will learn about how to use **decorators**, **Getters**, **Setters**, and **Deleters** in the class.

In a class, by adding **@property** to a method, we could make an instance from that method **without any arguments** just like an atribute. For example, we can call the *email()* method using the **Property** decorator like an atribute **email**.

[31]:
```
class Employee:


    def __init__(self, first, last, pay):

        self.first = first
        self.last = last
        self.pay = pay

    @property
    def email(self):
        return "{}.{}@gmail.com".format(self.first, self.last)


    @property
    def full_name(self):
        return "{} {}".format(self.first, self.last)

    # setter
    @full_name.setter
    def full_name(self, name):
        first, last = name.split(" ")
        self.first = first
        self.last = last

    # deleter
    @full_name.deleter
    def full_name(self):
        print("The name is deleted!")
        self.first = None
        self.last = None
```

```
    # special method __repr__()
    def __repr__(self):
        return 'Employee("{}", "{}", {})'.format(self.first, self.last, self.pay)

    # special method
    def __str__(self):
        return  "{}.{}".format(self.full_name, self.email)


empl = Employee("Sad", "Sikei", 50000)
empl_2 = Employee("Jahn", "Smith", 50000)
print("Please send my email:", empl.email, "to", empl_2.full_name)
```

```
Please send my email: Sad.Sikei@gmail.com to Jahn Smith
```

In **Setter**, we can assign a new argument to the *calss attributes*. For example, if we want to change the fullname of the emp1_2 by directly assigning *empl_2.full_name = "Hohn Jimmy"*, it throughs an AttributeError: "can't set attribute".

[ ]:
```
empl_2.full_name = "Hohn Jimmy"
print("Please send my email:", empl.email, "to", empl_2.full_name)
```

In **Deleter**, we can delete the attributes of the calss object.

[45]:
```
del empl_2.full_name
print(empl_2.full_name)
```

```
The name is deleted!
None None
```

In Addition to that, the special methods described with double under scores `__init__` are very important in the object oriented programming.

`__repr__` and `__str__` are the most commonly used special methodms in python object oriented programing.

`__repr__` is a special method used to represent a class's object as a string. `__repr__` is called by the `repr()` built-in function. One can define your own string represention of your class objects using the `__repr__` method. **Syntax:** `object.__repr__(self)`.

[23]:
```
empl = Employee("Sad", "Sikei", 50000)
print("This is a class object type: ", type(empl))
print(repr(empl))
#or
print("This is a string representation of the class object -- empl:", type(empl.
 ↪__repr__()))
```

```
This is a class object type:  <class '__main__.Employee'>
Employee("Sad", "Sikei", 50000)
```

```
This is a string representation of the class object -- empl: <class 'str'>
```

`__str__` is used for converting an object as a string, which makes the object more human-readable for end-users.

[32]:
```
empl = Employee("Sad", "Sikei", 50000)
print(str(empl))
```

```
Sad Sikei.Sad.Sikei@gmail.com
```

In addition to that, there are more about the **special methods** in Python. Special Methods