ООП Python

продолжение

```python
class MyClass:
    @classmethod
    def method(cls, arg):
        print('%s classmethod. %d' % (cls.__name__, arg))

    @classmethod
    def call_original_method(cls):
        cls.method(5)

    def call_class_method(self):
        self.method(10)

class MySubclass(MyClass):
    @classmethod
    def call_original_method(cls):
        cls.method(6)

# Вызываем методы класса через класс.
MyClass.method(0)  # MyClass classmethod. 0
MyClass.call_original_method()  # MyClass classmethod. 5

MySubclass.method(0)  # MySubclass classmethod. 0
MySubclass.call_original_method()  # MySubclass classmethod. 6

# Вызываем методы класса через объект.
my_obj = MyClass()
my_obj.method(1)  # MyClass classmethod. 1
my_obj.call_class_method()  # MyClass classmethod. 10
```

```
class Human:

    def __init__(self, name, age = 0):
        self.name = name
        self.age = age

    @staticmethod
    def is_age_valid(age):
        return 0 < age < 150

print(Human.is_age_valid(25))

human = Human("Nata", 23)
print(human.is_age_valid(23))
```

Может так получиться, что вам нужно объявить метод в контексте класса, но этот метод не оперирует ни ссылкой на конкретный экземпляр класса, ни самим классом непосредственно, как мы видели в методе класса. В таком случае вам может помочь статический метод.

```
class Robot:
    def __init__(self, power):
        self._power = power

    power = property()

    @power.setter
    def power(self, value):
        if value<0:
            self._power = 0
        else:
            self._power = value

    @power.getter
    def power(self):
        return self._power

    @power.deleter
    def power(self):
        print("make robot useless")
        del self._power



wall_e = Robot(100)
wall_e.power = -20
print(wall_e.power)
```

Property, или по-другому вычисляемые свойства. Зачем они нужны? \

Property позволяют изменять поведение и выполнять какую-то вычислительную работу при обращении к атрибуту экземпляра, либо при изменении атрибута, либо при его удалении.

```
class Robot:
    def __init__(self, power):
        self._power = power

    @property
    def power(self):
        return self._power


wall_e = Robot(100)
print(wall_e.power)
```

Иногда нужно как-то модифицировать чтение атрибута и выполнять какую-то полезную работу при чтении, и это единственное, что вам требуется. То есть не нужно менять поведение при изменении значения атрибута либо при его удалении. В таком случае есть более короткая запись. Мы можем объявить метод, обернуть его декоратором property без всяких суффиксов getter, setter и deleter, и это будет вычисляемым свойством класса

```python
class Pet:
    def __init__(self, name = None):
        self.name = name

class Dog(Pet):
    def __init__(self, name, breed = None):
        super().__init__(name)
        self.breed = breed
    def say(self):
        return f"{self.name}: waw"


dog = Dog("Sharik", "Pudel")

print(dog.name)
print(dog.say())
```

```python
#множественное наследование

import json

class Pet:
    def __init__(self, name = None):
        self.name = name

class Dog(Pet):
    def __init__(self, name, breed = None):
        super().__init__(name)
        self.breed = breed
    def say(self):
        return f"{self.name}: waw"


class ExportJSON:
    def to_json(self):
        return json.dumps({"name": self.name, "breed": self.breed})

class ExDog(Dog, ExportJSON):
    pass

dog = ExDog("Sharik", breed = "Pudel")
print(dog.to_json())
```